Adaptive Instruction Scheduling for the Itanium

Benoit Hudson Stephen Somogyi Shobha Venkataraman

Abstract

Many problems that compilers want to solve are NP-Hard to solve optimally. Therefore, compilers authors typically use fast ad-hoc methods that strike a trade-off between good code quality and fast compile time. We investigated taking the decision on the trade-off out of the compiler author's hands: based on profile information or static estimates, the compiler will tilt the balance in favour of code quality on hot segments of code, compensating by going for fast compile time on cold segments. The particular problem we decided to attack was instruction scheduling on the Itanium. We find inconclusive results, but we have an excuse: our machine model was inaccurate.

1 Introduction

In several places, optimizing compilers want to solve NP-hard optimization problems. Typically, this is done by having the compiler writer decide on a heuristic approach that, in practice, gives good results (outputs fast code), while still compiling the code in a reasonable amount of time.

Our goal is to allow trading off time and code quality at compile time. That is, the compiler itself may decide to spend longer – and get better code – on one part of the program than on another. This allows two things: the user can decide to compile quickly during the development phase, and spend a long time to get very good code for deployment; and the compiler can use profile feedback to identify hot sections and optimize them well, while saving time on less critical sections of code.

Two approaches that handle this notion well are anytime algorithms from the AI community, and PTAS from the algorithms / operations research community. In an anytime algorithm, the algorithm quickly finds a feasible solution, then conducts a search to find a better solution. When it finds a better solution, it stores the solution, then resumes searching. At any time (hence the name), the algorithm can be stopped and asked for its best solution so far.

A PTAS (polynomial-time approximation scheme) provides the user with a knob to turn: the algorithm will give an answer provably within $1+\epsilon$ of the optimum value, at a time cost of $O(n^{c+1/\epsilon})$ where c depends on the problem. For example, if c were 0, we might get a 2-approximation – the solution would provably be within a factor of two of optimal – in linear time, and a 1.1-approximation in $O(n^{10})$ time. Many, but not all, NP-hard optimization problems admit a PTAS.

1.1 Instruction scheduling

The problem we decided to study was scheduling instructions within a basic block, targeting the IA-64 architecture (specifically, the Itanium processor) [4]. In Section 2, we discuss how we infer that this problem is NP-Hard.

In basic block scheduling, the input is a set of instructions, a dependence graph between instructions, and a description of the processor. We draw an arc from instruction i_1 to instruction i_2

if i_2 must be scheduled after i_1. Furthermore, we put a weight on the arc to denote how long we must wait after i_1 has been issued before issuing i_2. An arc may have zero length, in which case i_2 can be issued at the same time as i_1 but not before (this is often the case with anti and output dependence). The task is to find, for each instructions, the time at which it should be issued such that the time at which the last instruction finishes is minimized.

The description of the processor typically includes the *issue width* – the number of instructions that can be simultaneously started – and a set of functional units. An instruction can only be issued on a certain subset of the functional units: for instance, an add instruction cannot be processed by a floating-point unit or a branch unit. We cannot issue more instructions at the same time-step than the issue width allows, and we cannot issue more than one instruction per functional unit.

On IA-64, we have additional constraints. The compiler must output code as a list of 128-bit *bundles* of three instructions. Furthermore, the bundle must match a *template*: for instance, we can issue a memory instruction followed by two integer instructions, but we cannot issue a memory instruction followed by two floating-point instructions. Some templates have *stops* between instructions that delimit the end of a region of instructions that the processor may issue concurrently. We discuss the highlights of the Itanium processor and the constraints posed by the design of the architecture, more extensively in Section 3.

Table of contents:

2 Related Work

Scheduling in the large is a well-studied problem and many books (indeed, theses) have been written about various variants of scheduling. In the nomenclature of the field [1, 3], instruction scheduling is closely related to a problem with parallel machines, precedence constraints, and where the objective is to minimize the makespan. This we denote by the cryptic string $P|prec|C_{\max}$.

Parallel machines (P) means that we have multiple machines, which correspond to functional units. Each job (instruction) has a time $p_{i\mu}$ which is the amount of time that job i spends on machine μ. We can say that a job cannot use a particular machine by setting $p_{i\mu}$ to be very large.

Precedence constraints (*prec*) very directly map to the data precedence graph.

The makespan ($C_{\max}$) is the time at which the last job finishes. This is exactly what we want.

We only say "closely related" because the nomenclature does not include an easy model for pipelining using parallel machines. That is, when we schedule instruction i on machine μ, machine μ is unavailable until instruction i is done. This is appropriate for non-pipelined machines but misses the pipeline entirely.

Another closely related problem is the job-shop problem $J|prec|C_{\max}$. Here a "job" i consists of a list of operations O_{ij} which must be done in series. Thus we can model each stage of a pipelined functional unit as a machine, and an instruction that requires that functional unit will have one operation per stage.

The job-shop model is much closer, but still not perfect. In particular, there is no way to specify

that an operation must be performed in the next cycle after the previous operation was performed. For example, on most machines, one instruction cannot stop processing after the third stage of the pipeline, let another instruction go through the pipeline, then resume processing in the fourth stage. But the job-shop model allows this.

This all implies that instruction scheduling on a normal pipelined superscalar machine is at least as hard as $P|prec|C_{\max}$ or as $J|prec|C_{\max}$. But Lenstra et al [8, 9] showed that even $P2||C_{\max}$ is NP-Hard. That is, no polynomial-time algorithm exists to find the shortest schedule when there are two machines and no precedence constraints – clearly an easier problem than instruction scheduling. Furthermore, the job-shop scheduling even without precedence constraints cannot be approximated better than 5/4 in polynomial time (assuming $P \neq NP$). That is, no polynomial-time algorithm can guarantee to produce a schedule that is less than 25% longer than optimal.

In light of these results, we can make the following two assertions:

- Scheduling on the IA-64 is NP-hard
- No PTAS exists for the problem.

Notice we say IA-64 rather than Itanium. This is because the hardness results hold when the machine description is an input to the algorithm. It is conceivable (though, in our opinion, highly unlikely) that due to the fact that the Itanium has exactly 9 functional units and issues instructions in precisely the fashion described by Intel, there is a PTAS or even a polynomial-time algorithm for scheduling. Consider that one could build a machine that implements the IA-64 architecture but is neither pipelined nor superscalar and all instructions take one cycle. On such a machine, finding the optimal schedule is trivial: any legal schedule is optimal.

However, the Itanium is not such a machine, thus it seems exceedingly likely that in fact scheduling n instructions on it is NP-hard and cannot be approximated better than a constant factor.

3 The Itanium Processor

The Itanium processor is the first of Intel's Itanium Processor Family, based on the IA-64 ISA (instruction set architecture). In particular, this ISA gives the compiler considerable control over the processor's behavior - thus it is called an Explicitly Parallel Instruction Computer (EPIC). EPIC was heavily influenced by VLIW (very long instruction word) computing, which is often beneficial for parallel/vector computation. EPIC takes this idea to the next level, and allows the compiler to identify independent instructions of all types - not just arithmetic operations.

To accomplish this, IA-64 uses the notion of *instructions groups*, which by definition are a set of independent instructions. The processor may rearrange the instructions in an instruction group in any order, and the compiler is responsible for ensuring correct output will always be generated.

IA-64 retains the concept of instructions, but introduces a new one: *bundles*. For the sake of efficiency, IA-64 operations are encoded as a bundle. A bundle is 128 bits, composed of three 41-bit instructions and a 5-bit template field. The template tells the processor what *kind* of instructions are present. The processor uses this information to deliver instructions to appropriate functional units.

To further confuse matters, there is not a direct correspondence between instruction type and functional unit type. There are five instruction types: I (integer), F (floating point), M (memory), B (branch), and L+X (used to encode longer immediate values). There are four types of functional unit: integer, memory, floating point, and branch. These appear to match fairly well initially,

however, it turns out that some instructions can be executed on either a I or a M unit (these are referred to as A-type instructions in some places in the literature, but are ignored in others).

As the template is only five bits, it cannot encode all possible bundle types. This is a very strict limitation than cannot be ignored by the compiler. Furthermore, the template encodes *where* instruction groups end. Thus, for a particular combination of instruction types, there are several other possibilities that demarcate different groups. For example, the MMI bundle can take one of four forms: MMI, $MMI \parallel$, $M \parallel MI$, $M \parallel MI \parallel$. Each "$\parallel$" represents a *stop*, which is Intel's name for the method of marking the end of an instruction group.

As stated above, the Itanium processor is the first implementation of the IA-64 architecture. It has 128 integer and 128 floating-point registers (the minimum dictated by the ISA), and is an in-order superscalar processor. It has two integer, two floating point, two memory, and three branch units. However, only one of each type of unit is fully capable, and the others are more restricted in exactly which instructions they can execute.

The Itanium processor will try to execute two bundles (i.e. six instructions) per clock cycle. Instructions must be issued in order (i.e. instruction #2 will not issue before instruction #1, regardless of stops). If an instruction cannot be issued because of resource limitations, the processor stalls all remaining instructions. If the first bundle is completely issued, a new bundle is brought in. In one clock cycle, the Itanium will only issue instructions from a single instruction group - i.e., a stop will always cause execution to stall until the next cycle, unless the stop occurs at the end of a bundle (when it would have to wait for the next cycle anyway). The Itanium processor also imposes further restrictions on which functional unit an instruction will be issued to based on the instruction's location within the bundle.

Obviously, all these complications must be considered if a compiler is to generate an optimal schedule. However, incorporating all of them into a clean problem model proved to be difficult. We consider a simplified model that takes into account some version of most constraints, and we do optimal scheduling in this model. We describe this model in Section 5, and in Section 5.3 we mention some of the major differences with respect to the true architecture.

4 Infrastructure

The Open Research Compiler (ORC) [7] is a tool for evaluating compiler advances for the Itanium Processor Family. As an open source project, the entire source code is available, permitting any optimization/improvement to be assessed. The current version of the compiler(2.1) is reasonably advanced, with support for all basic optimizations (e.g., peephole, dead code elimination) as well as some more advanced techniques (e.g., loop invariant code motion, inter-procedural optimization). The compiler also takes makes use of many of the advanced features of the IA-64 architecture, supporting if-conversion, predication, and control and data speculation.

With respect to instruction scheduling, ORC has two key features. First, scheduling occurs in two phases - global scheduling before register allocation, and local scheduling after that. Local scheduling is performed within a single basic block. Global scheduling occurs over regions, which are series of basic blocks with one entry point but multiple exit points. The compiler determines regions through code analysis and profiling information.

Profiling information is the second key feature to ORC's scheduling. Support for profiling was present throughout the compiler's development, and is integrated well. Profiling is also a two-phase process. First, the compiler instruments the binary, and when run, the program generates profile information. The compiler then re-instruments the program, ensuring that profile feedback is accurate in the presence of optimizations. A third compilation generates the final executable,

using the updated profile information.

5 Algorithm Description

As we discussed in Section 1 and Section 2, we had originally planned to use a PTAS to solve the instruction scheduling problem for Itanium processors, but we found no PTAS. Therefore we formulated a branch and bound any-time algorithm for the problem, based on the original list scheduling algorithm [10], modified to handle bundle constraints.

More formally, our model is the following: we have a set of machines M; a set of bundle templates B, each of which is a vector of three elements of M; and a set of instructions I. Each instruction i that may execute on a subset of the machines. Instructions may have dependencies on prior instructions (the dependency graph must be acyclic). With each dependency we associate a positive integer latency. Finally, we have a time bound in seconds which limits the amount of search we do: once the time is up, we return the solution we have (if we have no solution, we revert to list scheduling to produce one).

In this model, where latencies are associated not with instructions but with arcs between instructions, minimizing the completion time of the last instruction is ill-defined. Therefore we declare an instruction finished after it has spent the maximum latency of any of its dependents. We wish to find a schedule that minimizes the total time to complete the execution of the set I. In this model, that optimization criterion is equivalent to minimizing the start time of the last instruction to start.

We note that our model is a significant simplification of the true behaviour of the architecture. In particular, the bundling constraints and instruction issue constraints of the Itanium cannot be expressed by a mere set of bundle templates, especially constraining them to be sets of three machines. This issue is described in more detail in Section 5.3. It is a simple matter of programming to properly model the machine behaviour: we simply need to replace the word "triple" by the term "instruction group" and change how we check bundle constraints to match all the intricacies of the processor. Sadly, simple does not translate to being quick to do.

5.1 Branch and Bound Algorithm

We now describe our branch and bound algorithm for the simplified model of the scheduling problem.

First, we introduce the notation used in describing the algorithm. An *instruction group* is a set of three instructions, all ready, possibly including no-ops, that are scheduled into one bundle. R is list of instructions that are ready to be scheduled, F is the set of instructions currently in-flight, S is the set of instructions that are yet to finish (i.e., all instructions in flight, on the ready list, and not yet ready), and T is table of scheduled instructions, indexed by functional unit and time.

Figure 1 describes the basic list-scheduling algorithm that obey the bundle constraints. Here is how we pick triples of ready operations: we pick an instruction, then we pick a bundle template that allows that instruction, and then find two more ready instructions (or no-ops) that fit this template.

The branch and bound algorithm is essentially a modification of this algorithm. At the highest level, in the basic list scheduling algorithm, there are a few places where we simply choose the first valid option we come to. To get an optimal schedule, we instead need to loop over all choices (modulo pruning) at these points:

- deciding which instruction to process first from the ready queue.

```
SCHEDULE(R, F, S, T, time)
    /* if all instructions are scheduled and completed */
    while S is not empty
        time = time + 1
        /* update the sets of ready, unfinished, and in-flight instructions */
        S ← S − finished(F, time)
        F ← F − finished(F, time)
        R ← R + ready(F, time)
        /* find a valid set of ready instructions to schedule */
        find a triple z of ready operations we can schedule obeying bundle and machine
          constraints
        T ← T + {time, z}
```

Figure 1: The simple list-scheduling algorithm.

```
SCHEDULE(R, F, S, T, time)
    /* if all instructions are scheduled and completed */
    If S is empty
        if time < best-time,
            /* store the best schedule */
            best-schedule = T
            best-time = time
    else
        time = time + 1
        /* update the sets of ready, unfinished, and in-flight instructions */
        S' ← S − finished(F, time)
        F' ← F − finished(F, time)
        R' ← R + ready(F, time)
        /* try scheduling each valid set of ready instructions */
        for each triple z of ready operations we can schedule,
            T' ← T + {time, z}
            SCHEDULE(R' − z, F' + z, S', T', time)
```

Figure 2: The vanilla branch-and-bound algorithm.

- picking a bundle template.
- picking the additional ready instructions to fill out the bundle.

Figure 2 presents the branch and bound algorithm.

We observe that because bundle constraints only matter within a time step, we need to examine those only when finding valid instruction group.

5.2 Optimizing the algorithm

We now discuss some of the major optimizations we implemented on the vanilla branch and bound algorithm, to make it reasonably fast.

Data Structures The algorithm presented above is very sloppy about what data structures it uses and how it replicates them. To produce a fast (or at least bearable) implementation, we needed to be much more careful. Since our code was in C++, we used the data structures provided by the

STL. To avoid problems, we did not use the `hash_set` and `hash_map` types which were inexplicably left out of the C++ standard.

To tell whether an instruction is ready or finished, we need to be able to find when (and whether) its successors were started. Therefore, we store T as a `map` from an instruction to its start time. Interpret $T' \leftarrow T + \{time, z\}$ as being three insert calls.

A major cost of the algorithm is in replicating the data structures. In particular, S and T are of size linear in the number of instructions being scheduled. Since we schedule at most a constant number of instructions at each timestep, the search depth of the algorithm is itself linear. Therefore, we store a quadratic amount of information. Furthermore, each recursive call (each node in the search tree) does linear work copying S and T.

S, F, and R hold a lot of information, all of which can be inferred from T. However, we want to find that information more quickly than by looping over T. We can do away with S, which is the largest of the three, replacing the test for S being empty by a test for $F \cup R$ being empty: if there are no instructions in flight, and none ready, then we're done scheduling.

We could also avoid replicating all of T and instead change T in place: add z before the recursive invocation and remove it after. We did not do this because of time constraints.

We also make one final modification to the algorithm. Instead of storing the instructions that are in flight, we can store the *fringe*: the instructions waiting on instructions in flight (that is, all their predecessors are either in flight or have completed). By keeping the fringe ordered by when each instruction becomes ready, we can very quickly – that is, in time $O(|R'|)$ – update the ready list at the start of each timestep. Furthermore, we can skip a timestep if no instruction is ready and none becomes ready in that timestep.

Pruning Any speed-up in processing one node in the search tree is fairly minor: there are about $n!$ nodes to search in the tree (that is, schedules), so even if each node took constant time irrespective of the number of instructions to schedule, things would be painfully slow.

First, we observe that any time a group of instructions can be scheduled together, it will be at least as good as scheduling subsets of that group of instructions at that particular time. Therefore, we track if a set of instructions can be scheduled with additional instructions from the current ready list; if they can, we do not try to schedule that set of instructions.

Secondly, we note that in our model, it does not matter which bundle we choose – only that there exists a bundle for the triple of instructions –, since there are no interactions between bundles. In a truer model, we could not ignore this choice point.

Finally, we have so far not explained why the technique we used is called branch-and-bound. Notice that in our algorithm, the first thing we do is compute the list scheduling solution. From now on, if we can prove that any solution from the current state takes longer than the best solution known so far, we can immediately stop processing the current state. In particular, we need not bother making the recursive calls.

To determine whether we can prune, then, we need to quickly compute a lower bound for the remaining time in the schedule. For reasons known only to AI researchers, such a function is called an *admissible* heuristic.

For our heuristic we run a pre-processing phase in which we find, for each instruction, the longest path from that instruction in the precedence graph. Since the precedence graph is a DAG, the longest path is well-defined. The longest path is a lower bound for how long it will take to finish the schedule after scheduling an instruction because bundle constraints, issue width, and other considerations can only make the schedule longer.

As our heuristic value we consider the instructions in the ready list and in the fringe, and for

each we compute a lower bound on the finish time, taking into account that operations on the fringe cannot be issued yet. We return the maximum finish time over all these instructions: each finish time is a lower bound, so in particular, the largest is a lower bound (and it allows us to prune more).

This heuristic speeds up the search dramatically at no cost in correctness. We have not developed any other heuristics, but if we did, we could compute all the heuristics, take the largest of them, and use that as our lower bound. Or, if some heuristics were more expensive to calculate than others, we could compute the cheap ones first, and only use the expensive ones if the cheap ones were insufficient to prune.

Order Heuristics Since we prune based on the best solution found so far, it helps to find a good solution early on. Therefore, we order the instructions in the ready list so that we will go down promising branches early, in the hope that unpromising ones will be ignored long enough that we will be able to prune them.

Here, we need not use an admissible heuristic: any number will do. Only time is affected by the order in which we look at the branches of the search tree; correctness is not. We could even order the branches randomly – and in fact this is often useful to do.

However, the heuristic value we computed above does put higher value on instructions more likely to be a bottleneck, so we use it as our order heuristic. This means we behave like a textbook list scheduler to find the first solution. In experiments, we've found that a prioritized list scheduler does better than a list scheduler that picks instructions in arbitrary order; therefore, we believe that the order heuristic actually does speed up the search.

5.3 Simplifications: how our model is incorrect

Perhaps the biggest of the simplifications of our model is in the bundles. The Itanium documentation [5, 6] describes bundles as triples of instruction-types that may execute simultaneously, along with designated stop positions. We translate this into templates of machines by considering all combinations of machine types that execute each instruction-type in the bundle (modulo associativity), and ignore the stops. We do not represent any additional functional unit constraints.

This abstraction ignores any inter-bundle machine constraints. In reality, however, there are many inter-bundle machine constraints, and it is not just enough to examine the total number of machines available at any particular time-step; the machine on which an instruction is issued depends on the position of the instruction at that particular time, and whether it is the first or the second bundle. For instance, only machine *I1* may be used for an integer operation at the third instruction of the second bundle issued at a particular time. If we have two consecutive bundles b_1 and b_2, such that the last instruction i in b_2 is an integer instruction, and the machine *I1* is not available for i in b_2, the schedule splits the bundle b_2. However, if the bundle b_2 is split there, then that instruction i becomes the first instruction in the next time-step, and thus would need to be scheduled on machine *I0*.

A related simplification is that our model issues no more than 3 instructions at a time-step, rather than the 6 done by the Itanium – that is, only one bundle is issued at any time step. Once again, we did this because we do not represent inter-bundle machine constraints, but issuing up to 6 instructions at a time would require that we examine inter-bundle machine constraints, as well as track the stops in the bundles. That would be necessary to figure out when/where a bundle splits, so that we can figure out when a bundle is the first bundle in the instruction group, and when it is the second bundle. To illustrate this in our previous example, if the bundle template for

the second bundle b_2 was MMI, and there is more than one memory functional unit available for b_2, the bundle b_2 may be split only after the first M instruction. Therefore, there would only be one memory functional unit available for a new bundle b_3 brought in after b_1 finishes in the current time step. Thus the bundle b_3 may need to be split as well – if that happens, the bundle b_3 would become the first bundle in the time step after that, and will choose different machines, and so on.

Thus, we would need to track when a particular bundle may be brought in, as well as what machines are free at any particular time, and where a bundle can break. That would be the only way to figure out how to issue 6 instructions at a time, optimally, predicting how the bundles might split, and take into account the inter-bundle machine constraints. Also, in our current model, by only issuing one bundle at a time, we also can ignore the limitations on the number of machines, since any particular bundle has a set of instruction-types that can be scheduled together completely. In Section 5.4, we discuss the ORC's microscheduler and issues with using it to examine what sets of instructions may be scheduled at a particular time.

Also, there are instruction types (L and X) which execute on different machine types when the instructions have certain opcodes. We also ignore such constraints on the execution of instruction types, and assume that all instructions of an instruction-type are equivalent, and may be issued on the same set of machine-types. For the bundles that involved these instruction-types, we created all possible combinations of the machine-types involved.

5.4 Microscheduler

As described in Section 1.1, the characteristics of instruction issue on the Itanium are complicated, even though the Itanium is an in-order processor. Our initial understanding was that the processor would issue all three instructions in a bundle simultaneously, and that it would issue two bundles simultaneously if there were no data dependences between them. But this is false. Furthermore, in a paper by some of the developers of ORC [2], the authors find a 4.5% speed-up on average between an approach that largely ignored bundling constraints and a version that employed an FSA to model the state of the instruction issue.

ORC separates the task of scheduling into two parts: the high-level scheduler, which decides which instructions can be run simultaneously and tries to schedule the highest-priority ones first; and the low-level or microscheduler, which tries to reorder the instructions within an instruction group such that they will be run in parallel. The interaction is as follows: the high-level scheduler repeatedly picks an instruction it would like to schedule in the current cycle. The microscheduler then checks to see whether there is a template that allows issuing the instruction in the current bundle, which may require some reordering within the current and previous bundle due to instruction issue rules (the first integer instruction issued in a CPU cycle always goes to I0, for instance). If it cannot find one, the high-level scheduler tries another instruction. When the cycle is full, or if the microscheduler refused all the instructions, the high-level scheduler moves to the next cycle and commits the current bundle.

This is very much akin to the usual list scheduling. There, we can think of a low-level scheduler merely checking whether there is an open machine for the new instruction in the current cycle; the main difference is that the ORC microscheduler has a rather more complicated task than merely finding a machine.

5.4.1 Our experiences with ORC's microscheduler

We tried to get their microscheduler to work for us (rather than simply re-implementing it), but we failed. The problem is that they wrote their microscheduler assuming it would be run forwards

exactly once per basic block, whereas we want to backtrack. Accordingly, they only allow a 1-instruction look-ahead; after that, they start clobbering data structures throughout the back-end. That is, we can ask whether an instruction will fit in the current cycle, but we cannot even ask whether a set of instructions will all fit in the current cycle, much less decide on a set of instructions for the current cycle, advance to the next cycle, then restore to a prior state.

The ORC microscheduler is also not optimal, which is a problem when looking for an optimal schedule. It is, however, much better than our abstraction.

We do use the microscheduler in one way: The way we integrated our scheduler into ORC was by making our scheduler be the "heuristic" that the ORC scheduler uses to prioritize. This often ends up producing code that fits in far fewer cycles than what we predicted.

5.5 Memoization

In many search algorithms, the algorithm visits the same state repeatedly. To speed up the search, we can store a memo with each state saying what the optimal solution is from that state. When we enter a state, we first check to see whether there's a memo, and we use the memo if it exists rather than completing the search.

Memoization works well when the the number of states we actually visit is relatively small, and when we visit those same states often. If we only visited each state exactly once, there would be no win at all (instead, there would be a loss since making the memos and looking them up takes time).

To apply memoization to our problem, we need to define what a state in the search space is. One easy definition is that the state is the times of every instruction scheduled and the set of instructions not scheduled (the latter is implicit from the former, so we need not store it). However, under that definition we never repeat states in the algorithm above. Therefore, we use a different definition: the state is:

1. the instructions in the fringe and their start times compared to the current time, and
2. the set of instructions not yet scheduled.

Similarly, the solution we store will be the start times of the instructions that define the state (those in the fringe plus those not yet scheduled).

Using times relative to the current time, we can find some repetition. Consider the following situation: i_3 depends on both i_1 and i_2. Furthermore, i_1 and i_2 cannot be run in parallel, and there are more instructions after i_3. Let all latencies be one cycle. Suppose we start searching, and decided to run i_1 first, then i_2. The state at this point has a fringe of i_3 with a relative time of $+1$, and all instructions other than i_1 and i_2 are yet to be scheduled. We find an optimal solution for the remaining instructions – this may take a long time – and store it in a memo. Now the search will backtrack, and will try running i_2 before i_1 to see whether we can get a better schedule this way. Immediately, we will find the memo, and we will know that in fact no better schedule exists. Just this one memo will have cut down on the search by about half.

Admittedly, that was a contrived example. For a lack of time, we did not integrate the memoization code that we wrote into the search algorithm to test how useful it would be. We put a low precedence on this feature because it's not unlikely that the large size of the state would obviate any benefit it gave us: the state is of size up to linear in the size of the basic block.

6 Experimental Results

6.1 Experimental Setup

We evaluated our proposal with benchmarks compiled by the modified ORC. The benchmarks were run on a machine with an Itanium 1 processor at 733MHz and 1GB of RAM, running RedHat 7.3. A subset of the SPEC CPU2000 benchmark suite [11] was used - since the SPEC benchmarking process includes a verification stage, it ensures that our compiler generates correct code. Also, many researchers are familiar with the SPEC suite.

All benchmarks were run with profiling enabled. Our adaptive scheduling technique relies on profile information, and as such it would be an unfair comparison if the base case could not take advantage of the same information. Since ORC uses two profiling phases, three passes are required for each test case we are investigating - two passes with the training input for profiling, and one standard run with the reference input.

We compare three different scheduling techniques. The base case is the default ORC scheduler. Second, we use our base list scheduler. Third, we present results for our final branch-and-bound search algorithm. All compilation was performed at the highest optimization level (-O3) and with inter-procedural analysis enabled.

We use the profiling information to generate a reasonable time limit for the branch-and-bound scheduler. If a particular basic block accounts for much of the program's runtime, we should try longer to find the optimal solution. Our approach is to normalize the execution count of each basic block, by dividing its execution frequency by the sum for all basic blocks. This fraction is then multiplied by a constant to yield a value that is appropriate as a time limit (in seconds) for searching.

Our time constraints prevented the use of more benchmarks from the SPEC2000 suite. For each benchmark, nine runs must be performed: we test three scheduling algorithms, and each one takes three runs. Furthermore, the runs that collect profiling information run very slowly. For example, gzip takes as long with profiling on its training input as it does without profiling on its reference input.

We include results for the gzip and mcf integer benchmarks, and for the art, equake, and lucas floating point benchmarks. For reasons not clear to us, the list scheduler using our model failed to compile the equake benchmark (the compiler crashed).

6.2 Results and Discussion

The run times of the optimized executables running the reference inputs are shown in Figure 3. The standard ORC scheduler is usually faster than our schedulers, except for *art*, in which both of our scheduling techniques are clearly superior. Just between our two schedulers, there is no clear winner - they trade victory over different benchmarks.

The sizes in bytes of the compiled binaries are shown in the table in Figure 4 . Clearly, there is not much variation between the different scheduling techniques: our schedulers produce code that is about 0.1% smaller than ORC's code.

Accordingly, there is no relationship between code size and performance. Sometimes the smallest executable is fastest; other times it is slowest. Since the sizes are so similar, this result should not be surprising. Most code is not run very frequently, so its size can change without greatly affecting overall application performance

Relatively, our techniques perform considerably better with floating point applications than with integer applications. This is due to different types and frequencies of operations in the two

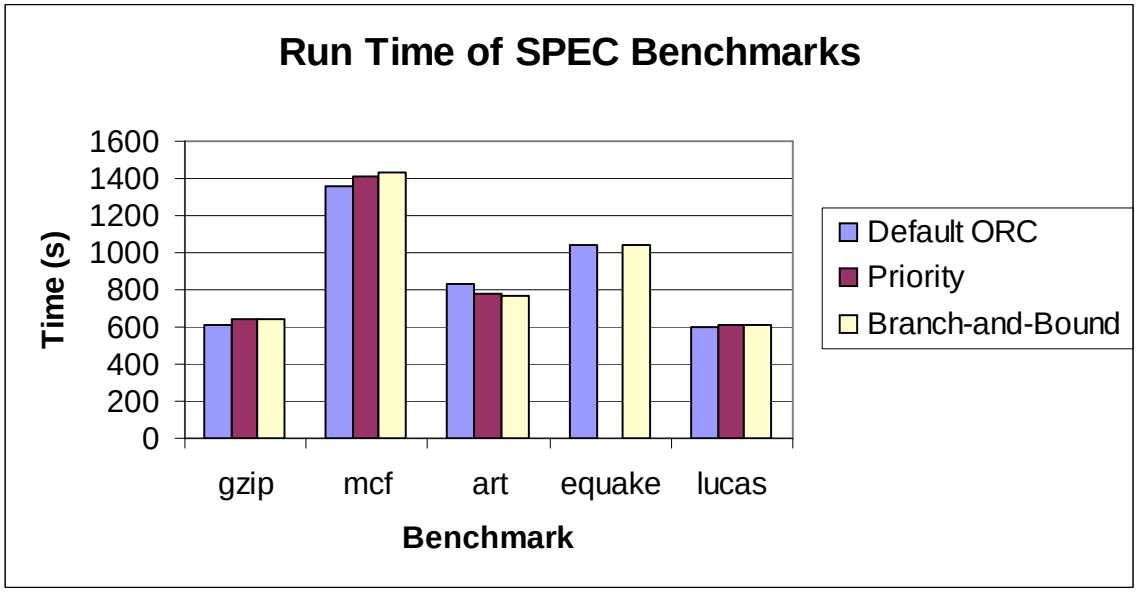

Figure 3: Graph showing the running times of the optimized executables from SPEC2000

Benchmark	Type	ORC	Priority List	Branch-and-Bound
gzip	int	156495	156039	155879
mcf	int	53283	53251	53307
art	fp	66835	66539	66667
equake	fp	83487	-	82895
lucas	fp	1721793	1721641	1721529

Figure 4: Sizes of the compiled binaries in bytes of various benchmarks of SPEC 2000

application classes. Floating point code is more regular, and instructions often have long latencies, which provide more opportunity for our schedulers to have an impact on performance. Conversely, integer code is a mess of low latency operations. Considering the complex restrictions that the Itanium processor uses to execute these instructions, a scheduler that has been tweaked for the Itanium will be able to find a better solution. This is the main purpose of ORC's microscheduler - to take care of these nuances. Since our scheduler does not take this level of detail into consideration, its code generation is not as good for integer applications.

It should be noted that our "optimal" scheduler sometimes does *worse* than the list scheduler. This is because the machine model is quite wrong.

7 Conclusion

We've mentioned that no PTAS is available (and likely none exists) for instruction scheduling on IA-64. However, this does not mean that the approach would not work for some other hard optimization problem in the compiler. Sadly, the ORC is ill-equipped for dealing with backtracking search algorithms, or any algorithm that changes its mind, making it hard to implement such algorithms in that framework. SUIF seems much better-equipped for such algorithms. We have no experience with other research compilers.

What we can conclude from our experience with the Itanium is that, as had been noted by other groups, the pipeline description, bundle constraints, and so on are critical to producing a good schedule. In fact, the slowdown we see compared to ORC is on the order of the slowdown they reported for using an overly naïve machine description, and better in some cases. This means that our experimental results are inconclusive.

Other essentially anecdotal evidence suggests that the branch-and-bound scheduler produces a significantly better schedule in our model than does the list scheduling algorithm. In the one program we tested in this way, the branch-and-bound scheduler produced basic blocks that (on a machine that matched our model) ran in 3.70 cycles on average, whereas our list scheduler produced code that ran in 4.41 cycles on average. This suggests that with a correct machine model, we should have beaten the ORC scheduler.

References

[1] Peter Brucker. *Scheduling Algorithms*. Springer Verlag, 1998.

[2] Dong-Yuan Chen, Lixia Liu, Chen Fu, Shuxin Yang, Chengyong Wu, and Roy Ju. Efficient resource management during instruction scheduling for the epic architecture. In *PACT'03*, pages 36–45, 2003.

[3] Leslie A. Hall. Approximation algorithms for scheduling. In *Approximation Algorithms for NP-Hard Problems*. PWS Publishing Company, 1995.

[4] J. Huck, D. Morris, J. Ross, A. Knies, H. Mulder, and R. Zahir. Introducing the ia-64 architecture. *IEEE Micro*, 20(5), September-October 2000.

[5] Intel. *Intel Itanium Processor Reference Manual for Software Optimization*, November 2001. http://www.intel.com/design/itanium/downloads/245474.htm.

[6] Intel. *Intel Itanium Architecture Software Developer's Manual. Vol 3: Instruction Set Reference*, October 2002. http://www.intel.com/design/itanium/downloads/245419.pdf.

[7] Intel open research compiler. Website: http://ipf-orc.sourceforge.net.

[8] J.K. Lenstra and A.H.G. Rinnooy Kan. The complexity of scheduling under precedence constraints. *Operations Research*, 26:22–35, 1978.

[9] J.K. Lenstra, A.H.G. Rinnooy Kan, and P. Brucker. Complexity of machine scheduling problems. *Annals of Discrete Mathematics*, 1:343–362, 1977.

[10] R.L.Graham. Bounds for certain multiprocessing anomalies. *Bell System Technical Journal*, 45:1563–1581, 1966.

[11] Spec 2000 cpu performance benchmarks. http://www.spec.org.