Application Insight Through Performance Modeling

Gabriel Marin
Department of Computer Science
Rice University
Houston, TX 77005
mgabi@cs.rice.edu

John Mellor-Crummey
Department of Computer Science
Rice University
Houston, TX 77005
johnmc@cs.rice.edu

Abstract

Tuning the performance of applications requires understanding the interactions between code and target architecture. This paper describes a performance modeling approach that not only makes accurate predictions about the behavior of an application on a target architecture for different inputs, but also provides guidance for tuning by highlighting the factors that limit performance in each section of a program. We introduce two new performance metrics that estimate the maximum gain expected from tuning different parts of an application, or from increasing the number of machine resources. We show how this metric helped identify a bottleneck in the ASCI Sweep3D benchmark where the lack of instruction-level parallelism limited performance. Transforming one frequently executed loop to ameliorate this bottleneck improved performance by 16% on an Itanium2 system.

1 Introduction

Accurate performance models of applications can be used to understand how their performance scales under different conditions, pinpoint sources of inefficiency, identify opportunities for tuning, guide mapping of application components to a collection of heterogeneous resources, or provide input into the design of architectures. Building accurate models of application performance is difficult because of the large number of variables that affect the execution time, including algorithmic factors, program implementation details, architecture characteristics, and input data parameters. Moreover, these factors interact in complex ways, making it difficult to understand what limits the performance for each section of the code. Hardware counters, present on all modern microprocessors, provide a low overhead solution for observing resource utilization during an application's execution. While they can provide insight on how resources (*e.g.*, time) are being expended for a given input size and target architecture, they cannot provide estimates of how execution time will change if we change the problem size or the type and number of resources in the architecture. Also, they cannot directly pinpoint program sections that will benefit most from code transformations.

In this paper, we describe an approach for measuring and modeling application characteristics independent of the underlying architecture. We use the resulting models to understand features of the application that limit its performance and we explore how the application maps onto a target architecture to identify resources that limit performance due to a high level of contention. We show that our application-centric modeling technique can provide insight into applications that hardware counter based tools alone cannot.

Our goal is to answer questions about an application's performance. Is the program slow because of a costly algorithm? Does the program have insufficient instruction-level parallelism? Is there a mismatch between the type and number of resources provided on the target machine and the type of resources required by the most frequently executed loops of the application? Is the application bandwidth starved, or is it limited by the memory latency? How much memory hierarchy bandwidth is needed to fully utilize the execution units if we could perfectly prefetch all data?

We introduce two new performance metrics that enable us to pinpoint sections of the application that have insufficient instruction-level parallelism or poor memory balance, and sections that have a high level of instruction parallelism and might benefit from an attached accelerator (*e.g.*, an FGPA). In addition, our modeling technique may be used to inform the design process for future architectures about which high-level architectural changes would most benefit the application.

The rest of the paper is organized as follows. Section 2 introduces our modeling framework and motivates our design decisions. Section 3 describes our approach for gaining insight into application performance bottlenecks. Section 4 presents case studies in which we use our models to analyze the ASCI Sweep3D benchmark and LANL's Parallel Ocean

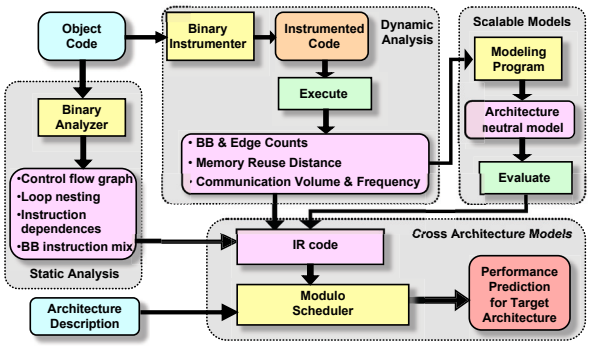

Figure 1. Modeling framework diagram.

Program (POP). Section 5 identifies related work in the area of performance modeling and prediction. Section 6 presents our conclusions and plans for future work.

2 The Modeling Framework

We use static and dynamic analysis of binaries as the basis for performance modeling and prediction. Because we analyze and instrument object code, our tools are language independent and naturally handle applications with modules written in different languages. In addition, binary analysis works equally well on optimized code; we do not need our own optimizer to predict performance of optimized code. Finally, it is easier to predict performance of a mix of machine instructions with predictable latencies than to estimate the execution cost of high-level language constructs. Working on binaries, however, has its own drawbacks. First, the tool's portability is limited by the binary analysis library it uses. This is mitigated by the fact that we measure and model application characteristics that are machine independent, and then we use these models to predict performance on arbitrary RISC-like architectures. Second, certain high-level information is lost when the source code is translated to low-level machine code, while other information requires more thorough analysis to extract.

Figure 1 shows an overview of our modeling framework. There are four main functional components: static analysis, dynamic analysis, an architecture neutral modeling tool, and a modulo-scheduler for cross-architecture performance prediction. Most of the components of our modeling framework are described in detail elsewhere [9]. In section 3, we describe our modulo-scheduler and other extensions that provide insight into applications and reveal performance bottlenecks. To provide context for that work, we briefly describe the components of our framework.

The *static analysis* subsystem shown in Figure 1 is not a standalone application but rather a component of every program in our toolkit. We employ static analysis to recover high-level program information from application binaries, including reconstruction of the control flow graph (CFG) for each routine and identifying natural loops and loop nesting in each CFG using interval analysis [15]. Other uses of static analysis involve understanding and modeling aspects of the application that are important for its performance but are independent of execution characteristics, *e.g.*, the instruction mix in loop bodies, as well as dependencies between instructions that limit instruction-level parallelism (ILP). Understanding memory dependencies within loops from machine code is a particularly difficult problem that we tackle by computing symbolic formulae that characterize the access pattern of each memory reference, a process that is described in more detail elsewhere [10].

Often, many important characteristics of an application's execution behavior can only be understood accurately by measuring them at run time using *dynamic analysis*. Our toolkit uses binary rewriting to augment an application to monitor and log information during execution. To understand the nature and volume of computation performed by an application for a particular program input, we collect histograms indicating the frequency with which particular control flow graph edges are traversed at run time. To understand an application's memory access patterns, we collect histograms of the reuse distance [4]—the number of unique memory locations accessed between a pair of accesses to a particular data item—observed by each load or store instruction. These measures quantify key characteristics of an application's behavior upon which performance depends. By design, these measures are independent of architectural details and can be used to predict the behavior of a program on an arbitrary RISC-like architecture.

Collecting dynamic data for large problem sizes can be expensive. To avoid this problem, we construct *scalable models* of dynamic application characteristics. These models, parameterized by problem size or other input parameters, enable us to predict application behavior and performance for data sizes that we have not measured [9, 10].

To compute *cross-architecture performance predictions*, we combine information gathered from static analysis and dynamic measurements of execution behavior. We then map this information onto an architectural model constructed from a machine description file. This process has two steps. First, we predict the program's memory hierarchy behavior by translating memory reuse distance models into predictions of cache miss counts for each level of the memory hierarchy on the target architecture. We predict capacity and compulsory misses directly from the reuse distance models and estimate conflict misses using a probabilistic strategy [10]. Second, we predict computation costs by identifying paths in each routine's CFG and their associated frequencies, and mapping instructions along these paths onto the target architecture's resources using a modulo instruction scheduler. This process is described in section 3.1.

3 Understanding Performance Bottlenecks

In our previous work [9, 10] we validated our approach by successfully predicting memory hierarchy behavior and execution time of several scientific codes, including the ASCI Sweep3D benchmark [1] and several NAS benchmarks, on different architectures for a large range of input sizes. In this section, we show how models can provide insight into applications and reveal performance bottlenecks. The cornerstone of this analysis is our modulo-scheduler, which maps a sequence of generic instructions with their corresponding data dependencies to a collection of heterogeneous resources, in accordance with a mapping function between instructions and resources, and with the objective function of minimizing the schedule length. The next sections describe the functionality and design of our scheduler.

3.1 Scheduler front end

To understand the cost of a program's computation on a target architecture, we start by identifying paths in the CFG of each routine and computing their associated frequencies from basic block and edge frequency counts collected during dynamic analysis. For nested loops, we work from the inside out, each loop being represented by a special Inner-Loop instruction in its parent scope. Once a loop is scheduled, we also compute information about its registers that are live across the loop boundaries. Such information is used to compute the proper register dependencies between a loop and instructions in its parent scope. In addition, inner loops and function calls act as fence instructions in our schedules; they prevent instruction reordering across them.

As seen in Figure 1, the scheduler works on an intermediate representation (IR) of the code. We found the most flexible format for the IR is a dependence graph, in which nodes represent generic instructions and edges represent dependences between instructions. Figure 2(b) shows the dependence graph for the innermost loop of routine `compute` shown in Figure 2(a).[1] Using such an intermediate representation has two main benefits. First, it isolates the scheduler from the binary analysis library underneath, making it portable to a different run-time system. We need to provide only a front end that translates machine code into the IR by identifying all schedule dependencies among instructions. Second, the scheduler can be used to analyze sequences that include higher-level operations (i.e. FFT or dot product operations) if the machine description provides units which can execute such instructions, and if the front end recognizes such operations and includes them as nodes in the IR.

In our implementation, we defined a set of generic RISC instruction classes and our front-end translates SPARC machine instructions into the intermediate representation. We defined a machine description language (MDL) that must be used to model the target architecture. The architecture description at a minimum must define the number and type of execution units, and the resources required by each generic instruction on that architecture. Instead of providing the entire grammar of our MDL, we illustrate the most important language constructs with snippets from our description of the Itanium2 architecture [8].

[1] To reduce clutter, the graph does not include the loop control arithmetic and the loop branch instruction.

```
void
compute(int size, double* A, double c1){
int i, j;
for (j=0 ; j<size ; ++j)
   for (i=0 ; i<size-1 ; i+=1){
      A[(i+1)*size+j] =
             A[(i+1)*size+j] +
             c1 * A[(i)*size+j];
   }
}
```

(a)

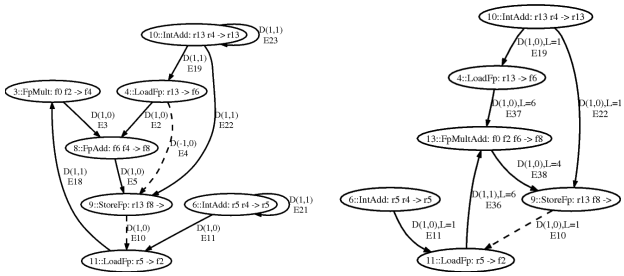

Figure 2. (a) Sample source code; (b) IR for the inner most loop; (c) IR after replacement rules and edge latencies are computed.

3.2 Machine description language

Figure 3 presents the language construct for defining the list of execution units (top) and an optional construct for defining special restrictions between units (bottom). When defining the available execution units, an optional multiplicity factor for each unit class can be included, if there are multiple units of the same type. Using the multiplicity operator simplifies the declaration of both the list of available units and of the instruction execution templates. In addition, it provides a single point of control when playing with alternative machine designs that have different number of units of a given type. Additional units can be declared just to simplify the definition of restrictions between different instruction execution templates within the constraints of the language. Notice that for the Itanium2 model we included

List of execution units (EU):

```
CpuUnits = U_Alu*6, U_Int*2, U_IShift,
           U_Mem*4, U_PAlu*6, U_PSMU*2,
           U_PMult, U_PopCnt, U_FMAC*2,
           U_FMisc*2, U_Br*3,
           I_M*4, I_I*2, I_F*2, I_B*3;
```

Special restrictions between EUs:

```
Maximum 1 from U_PMult, U_PopCnt;
Maximum 6 from I_M, I_I, I_F, I_B;
```

Figure 3. MDL constructs for defining the execution units and restrictions between units.

Instruction execution templates:

```
Instruction LoadFp template =
       I_M + U_Mem, NOTHING*5;
Instruction StoreFp template =
       U_Mem[2:3](1)+I_M[2:3](1);
```

Instruction replacement rules:

```
Replace FpMult $fX, $fY -> $fZ +
        FpAdd $fZ, $fT -> $fD with
   FpMultAdd $fX, $fY, $fT -> $fD;

Replace StoreFp $fX -> [$rY] +
        LoadGp [$rY] -> $rZ with
   GetF $fX -> $rZ;
```

Figure 4. MDL constructs for declaring instruction execution templates (top) and instruction replacement rules (bottom).

also a list of issue ports in addition to the execution units. Using the convention that each instruction template must declare the use of one issue port of proper type in addition to one or more execution units, we can restrict the number and type of instructions that can be issued in the same cycle. For example, to model the six issue width of the Itanium2 processor, the second restriction rule in Figure 3 specifies that at most six issue ports can be used in any given cycle.

Figure 4 presents examples of instruction execution templates and instruction replacement rule declarations. An instruction template defines the latency of an instruction, and the type and number of execution units used in each cycle. The first instruction template in Figure 4 represents the most common format of template declaration, thus the shortest. It applies to instructions that execute on fully pipelined symmetric execution units. On Itanium2, floating-point loads can be issued to any of the four memory units, and have a minimum latency of six cycles when data is found in the L2 cache. Thus, one `LoadFp` instruction is declared to need one issue port of type `I_M` and one execution unit of type `U_Mem` in the first clock cycle, plus five additional clock cycles in which it does not conflict with the issue of any instruction. *NOTHING* is a keyword which specifies that no execution unit is used. Instruction templates can make use of the multiplicity operator to specify consecutive clock cycles that require the same type and number of resources. The second template in Figure 4 shows the extended form of declaring an execution template which is needed in case of asymmetric execution units. While floating-point loads can execute on any of the four memory type units, stores can execute only on the last two units. Thus, this template uses the optional range operator in square brackets to specify a subset of units of a given type, and the count operator between round parentheses to specify how many units of that type are needed. Instructions can have associated multiple execution templates, possibly with different lengths.

Instruction replacement rules, presented in the bottom half of Figure 4, are an important type of language construct used to translate sequences of instructions from the instruction set of the input architecture, into functionally equivalent sequences of instructions found on the target architecture. We introduced the replacement construct to account for slight variations in the instruction set of different architectures. For example, the SPARC architecture does not have a multiply-add instruction, while Itanium2 does. Moving data between general-purpose and floating-point registers is accomplished on SPARC by a save followed by a load from the same stack location using registers of different types. Itanium2 provides two instructions for transferring the content of a floating-point register to a general-purpose register and back. In addition, the IA-64 instruction set does not include the following type of instructions: integer multiply, integer divide, floating-point divide, and floating-point square root. Floating-point divide and square root operations are executed in software using a sequence of fully pipelined instructions. The integer multiply and divide operations are executed by translating the operands to floating-point format, executing the equivalent floating-point operations, and finally transferring the result back into a fixed-point register. In our Itanium2 architecture description, we provide replacement rules for all these type of instructions. Figure 2(c) presents the dependence graph for loop `i` of the code shown in Figure 2(a) after the replacement rules were applied. One multiply and one add instructions were replaced with a single multiply-add.

The final two constructs are shown in Figure 5. Bypass latency rules are used to specify different latencies than what would normally result from the instruction execution templates for certain combinations of source instruction type, dependence type, and sink instruction type. The two bypass rules shown in Figure 5 refer to control dependences. For example the second rule specifies that a branch or function call instruction can be issued in the same cycle

Bypass rules:

```
Bypass latency 1 for ANY_INSTRUCTION
       -> [control] InnerLoop;
Bypass latency 0 for ANY_INSTRUCTION
       -> [control] CondBranch |
                    UncondBranch |
                    Jump;
```

List the Memory Hierarchy Levels (MHL):

```
MemoryHierarchy =
      L1D  [256, 64, 4, *, L2D, 4],
      L2D  [2048, 128, 8, 32, L3D, 8],
      L3D  [12288, 128, 6, 32, DRAM, 110],
      DRAM [*, 16384, *, 7, DISK, 10000],
      TLB  [128, 8, *, 1, L2D, 25];
```

Figure 5. Example of MDL constructs for defining bypass latency rules (top) and for describing the memory hierarchy levels (bottom).

as an instruction that precedes it if there are no other type of dependences between them, even if the source instruction normally has a long latency.

The last MDL construct in Figure 5 defines the characteristics of the memory hierarchy. For each memory level, the parameters are: number of blocks, block size (bytes), associativity, bandwidth on a higher level miss (bytes/clock), memory level accessed on a miss at this level, penalty in cycles for going to that level. The value of some attributes can be omitted and then a default value is used, depending on the attribute type.

3.3 Scheduler implementation

We implemented an architecture generic, critical-path driven, bidirectional modulo scheduler. It is close in concept to Huff's bidirectional scheduler [7], although we do not consider register pressure among the scheduling priorities at this time. The scheduler starts by pruning the dependence graph of edges that create trivial self-cycles, and other redundant edges. Next, the graph is transformed by performing replacement operations described in the machine description file, and all edges of the new graph are assigned a latency value based on the bypass latency rules and the instruction execution templates. Once the latencies are computed, the graph is pruned one more time, using the latency information to identify and remove trivial edges.

Once all dependences between instructions and their associated latencies are computed, the scheduler can compute the minimum initiation interval (MII), which represents the minimum schedule length that is theoretically achievable. The schedule length is bounded below by two factors: resource contention and dependence cycles. An execution unit can be in use by at most one instruction in any given clock cycle. The lower bound due to resource contention, LB_{Res}, is determined by how tightly we can map all instructions from one loop iteration to the machine execution units, if we assume no dependences between instructions.

$$LB_{Res} = \max_{u \in \mathcal{U}} (\text{uses}(u)),$$

where $\mathcal{U}$ is the set of available execution units and $uses(u)$ represents the number of clock cycles unit u is busy for the instructions in one loop iteration.

Separately, we compute a lower bound due to recurrences, LB_{Dep}. For this, we assume a machine with unlimited number of resources and the bound is determined by the longest dependence cycle. All graph edges have associated length and distance information. The length is given by the latency computed in a previous step. The distance is computed by the scheduler's front-end as part of its dependency analysis phase. Dependences can be either *loop independent* or *loop carried* [3]. Loop-independent dependences have both their ends in the same iteration and their distance is $D = 0$. For loop-carried dependences, the sink instruction depends on an instance of the source instruction from $d > 0$ iterations earlier, and the distance in this case is $D = d$. For the example in Figure 2(c), edge *E36* from the `LoadFp` instruction to the `FpMultAdd` instruction is the only loop-carried dependence and has a distance of 1. All other dependences are loop independent.

For each dependence cycle c, we compute the sum of latencies $L(c)$ and the sum of distances $T(c)$ over all its edges. Every recurrence must contain at least one carried dependence. As a result $T(c)$ is guaranteed to be strictly positive. If an instruction is part of a recurrence with total length $L(c)$ and total distance $T(c)$, then it can start executing no earlier than $L(c)$ clock cycles after its instance from $T(c)$ iterations earlier executed. Thus, each recurrence creates a lower bound on schedule length equal to $\lceil L(c)/T(c) \rceil$, and the lower bound due to application dependences is:

$$LB_{Dep} = \max_{c \in \mathcal{C}} \left\lceil \frac{L(c)}{T(c)} \right\rceil,$$

where $\mathcal{C}$ is the set of dependence cycles.

The minimum initiation interval becomes MII $= \max(LB_{Res}, LB_{Dep})$. In practice, most loops can be scheduled with a length equal to this lower bound. However, for some loops, accommodating both dependences and resource constraints increases the feasible schedule length. To find the schedule that can be achieved, we start with a schedule length k equal to MII and increase it until we can successfully map all instructions onto the available resources in k clock cycles. We map instructions one by

one, in an order determined by a dynamic priority function that tracks how much of each recurrence is still not scheduled. We use limited backtracking and unscheduling of operations already scheduled when the algorithm cannot continue. The full details on the implementation of this step are beyond the scope of this paper.

3.4 Performance analysis extensions

So far, we have described the steps of a fairly standard modulo-scheduling algorithm. Our implementation has its strengths and weaknesses. A strength of the scheduler is that it can be applied to different scheduling scenarios. A weakness of the scheduler is that it ignores some architectural details, such as register pressure or branch miss prediction. It is adequate for our purposes because we want to predict a lower bound (though not too loose) on achievable performance, and to understand what sections of code may benefit from transformations or from additional execution units. A modulo-scheduler offers us this insight because we can attribute each clock cycle of the resulting schedule to a particular cause.

When we compute the minimum initiation interval of the schedule, if $LB_{Dep} \geq LB_{Res}$, we consider that LB_{Dep} clock cycles of each loop iteration are due to *application dependences*. If, on the other hand, the bound due to resource contention is greater, we know also which unit was determined to have the highest contention factor.[2] If multiple units have the same contention factor, the first unit defined in the machine description file is selected. In such cases we say LB_{Res} clock cycles of each iteration are due to *resource bottlenecks*, and we refine this cost further by the type of unit that is causing the most contention.

In the next step of the algorithm, we try to find an actual feasible schedule length that takes into account both instruction dependences and resource contention. Every time we increase the schedule length, we determine what resource, either execution unit or restriction rule, prevented the scheduler from continuing. The way this scheduling step is implemented, the algorithm does not try to schedule an instruction in a clock cycle that breaks dependences. Therefore, the scheduler fails when there is no execution template that does not conflict with resources already allocated or with one of the optional restriction rules for any of the valid issue cycles. This additional scheduling cost for each iteration is counted separately as *scheduling extra cost*. Again, we refine this cost further by the unit type that was the source of contention.

We compute the execution costs in a bottom-up fashion, from the innermost loops to routines and to the entire program, aggregating costs for each of the categories described above. At the end of this process, we have not only a prediction of instruction schedule time for the entire program, each routine and each loop, but also the attribution of execution cost to the factors that contribute to that cost.

Performance monitoring hardware on most modern architectures can provide insight into resource consumption on current platforms. Our performance tool can provide such insight for future architectures, at a much lower cost than cycle accurate simulators. However, we realized that what is lacking in current performance tools, is a way to point the application developer, or an automatic tuning system for that matter, to those sections of code that can benefit the most from program transformations or from additional machine resources. Just because a loop accounts for a significant fraction of execution time, it may not be wasting issue slots. It may actually have good instruction and memory balance with respect to the target architecture with little room for improvement. We need to focus on loops that are frequently executed but also use resources inefficiently.

3.5 New performance metrics

One of the steps in the scheduling algorithm is the computation of the minimum initiation interval. For this, two lower bounds on the schedule length are computed. LB_{Res} represents the lower bound due to resource contention, and is computed assuming no schedule dependences between instruction. Let $\mathcal{S}$ be the computation cost computed by the scheduler when both instruction dependences and resource constraints are considered. We define the metric *maximum gain achievable from increased ILP* as $MaxGain_{ILP} = \mathcal{S} - LB_{Res}$. This metric represents exactly what its name implies. If total computation cost is $\mathcal{S}$, and the cost achievable with the same set of machine resources if we removed all data dependences in loops is LB_{Res}, then the maximum we can expect to gain from transforming the code to increase ILP, is $\mathcal{S} - LB_{Res}$. If the code's performance is limited by the number and type of machine resources, that is if $\mathcal{S} = LB_{Res}$, there is nothing that can be gained from transforming the code, unless we rewrite the code using different instructions that require different resources.

LB_{Dep} represents the lower bound due to dependence cycles, and is computed assuming unlimited number of machine resources. We define the metric *maximum gain achievable from additional resources* as $MaxGain_{Res} = \mathcal{S} - LB_{Dep}$. The name of this metric is self explanatory. No matter how many execution units we add to a machine, the execution cost of the code cannot be lower than LB_{Dep} unless we also apply code transformations. For loops without recurrences, LB_{Dep} of N iterations is equal to the execution cost of one iteration from start to finish, independent of N. With an unlimited number of machine resources, and

[2]Because the machine model may contain optional restriction rules between units, if one of the rules is determined to cause the most contention, then the cost is associated with that rule

with no carried dependences, all iterations can be executed in parallel in the time taken by a single iteration. However, we do not apply the same idea to outer loops or loops with function calls. As we explained in Section 3.2, inner loops and function calls act as fence instructions, so they create at least a recurrence on themselves.

Each of these two metrics gives an estimate of the performance increase possible by modifying only one variable of the equation in isolation. At an extreme, if we removed all dependences and we assumed an infinite number of resources, we could execute a program in one cycle. But this is too unrealistic to be of any use in practice. As with other performance data we compute, we aggregate these two metrics in a bottom-up fashion up to the entire program level. To explore this data, we output all metrics in XML format, and use the `hpcviewer` user interface [11] that is part of HPCToolkit [12].

4 Case Studies

In this section, we briefly illustrate how to analyze and tune an application using these new performance metrics. We study two applications. Sweep3D [1] is a 3D Cartesian geometry neutron transport code benchmark from the DOE's Accelerated Strategic Computing Initiative. As a benchmark, this code has been carefully tuned already, so the opportunities for improving performance are slim. The Parallel Ocean Program (POP) [2] is a climate modeling application developed at Los Alamos National Laboratory. This is a more complex code with the execution cost spread over a large number of routines. We compiled both codes on a Sun UltraSPARC-II system using the Sun WorkShop 6 update 2 FORTRAN 77 5.3 compiler, and the optimizations: *-xarch=v8plus -xO4 -depend -dalign -xtypemap=real:64*.

For Sweep3D, we used our toolkit to collect edge frequency data and memory reuse distance information for a cubic mesh size of 50x50x50 and 6 time steps without fixup code. For POP, we collected data for the default benchmark size input. Then, we processed collected data with our modulo-scheduler using a machine model of the Itanium2 architecture, and produced performance databases in XML format for each of the codes. In the next two sections we look at each code on turn.

4.1 Analysis of Sweep3D

Figure 6 shows a snapshot of the `hpcviewer` interface browsing the performance database sorted by the $MaxGain_{ILP}$ metric (in the bottom right pane). Due to limited horizontal space in a paper, only two metrics are shown: i) maximum gain expected from increased ILP, and ii) predicted computation time. We expanded six levels of

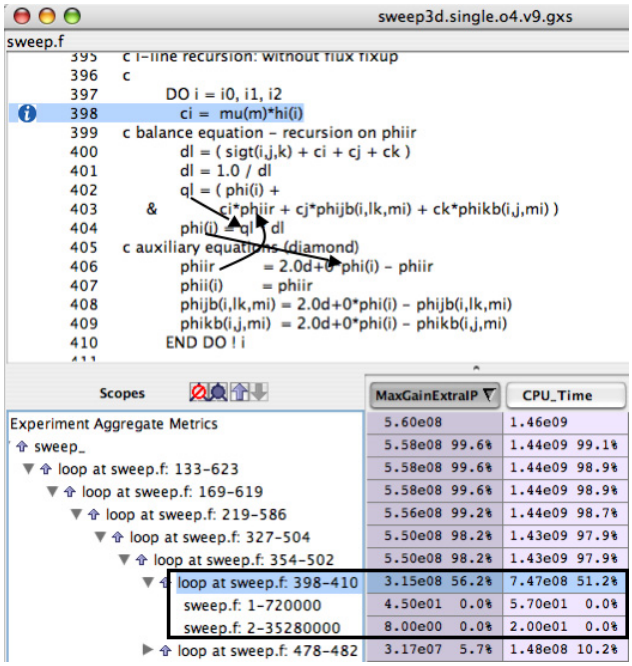

Figure 6. Sweep3D performance data view.

scopes that are shown to account for over 98% of the potential for improvement according to this metric. While 98% of this potential is contained within the level five loop at lines [354–502], the I-line recursion loop without flux fixup at lines [398–410] accounts for 56% of the entire potential and the next most significant inner loop accounts for only 5.7% of this potential. It is clear that we have to focus our attention on this loop. By expanding the scope of this loop, we expose the performance data for the two paths through this loop. While for loops we present metric totals, for paths we show metric values per iteration.

When we reconstruct the paths taken through a loop, we consider the paths that follow the loop back-edge and the exit paths separately. Back-edge paths are scheduled using software pipelining, while for exit paths software pipelining is disabled. For this loop, first path is the exit path and we notice the program enters this loop 720K times, and second path is the back-edge path which is executed over 35 million times. Looking at predicted computation time if we had an infinite number of execution units, metric not included in the figure, we notice that even if we had an infinitely wide machine the time per iteration would still be 20 clock cycles, while if we could remove the dependences, the time per iteration would drop by 40% to 12 clock cycles. This is the clear sign of a recurrence of 20 clock cycles in the code. We spotted two short recurrences, but the longest recurrence is the one marked in the source pane of Figure 6.

By manual inspection, we realized that loop `jkm` at lines [354–502] has no carried dependencies. If we unroll the `jkm` loop and then fuse all the inner loops, we can execute

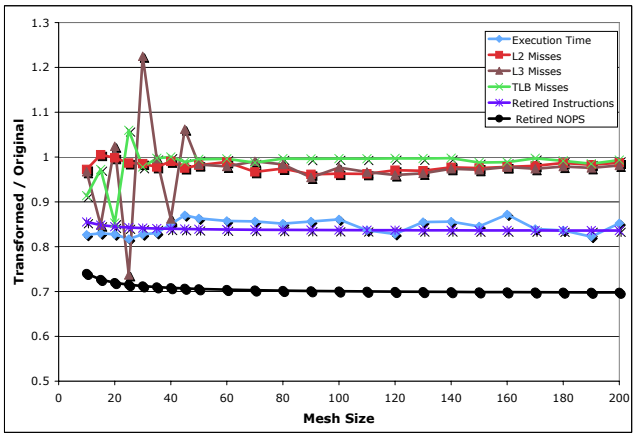

Figure 7. Performance of the transformed Sweep3D code relative to the original version on an Itanium2 (lower is better).

multiple I-line recursions in parallel, effectively increasing the ILP. We decided to unroll the `jkm` loop only once since we already fill 60% of the issue cycles with one iteration. After transforming the code, we ran it again through our tool. The predicted overall computation time[3] dropped by 20% from $1.46e09$ down to $1.17e09$, and the total potential for improvement from additional ILP has dropped from $5.60e08$ down to $1.35e08$. This potential, however, is due only to loop exit paths and outer loops that cannot be effectively pipelined. For the I-line recursion loop the value of this metric dropped by a factor of 25.

All numbers presented so far are predictions from our tool. To see if these predicted improvements can be observed on a real Itanium2 machine as well, we compiled both the original and the transformed Sweep3D codes on an Itanium2 based machine running at 900MHz. We compiled the codes with the Intel Fortran Itanium Compiler 9.0, and the optimization flags: *-O2 -tpp2 -fno-alias*.[4] Using hardware performance counters we measured the execution time, the number of L2, L3 and TLB misses, and the number of instructions and NOPs retired, for both binaries and for mesh sizes from $10\times10\times10$ to $200\times200\times200$. Figure 7 presents the performance of the transformed code relative to the performance of the original code for all input sizes. We notice the transformed program is consistently faster by 13-18% with an average reduction of the execution time of 15.6% across these input sizes. The number of cache and TLB misses in the two versions of the code differ by only 2-3%, thus they cannot account for the performance increase. The spikes for L3 and TLB misses at small problem sizes are just an effect of the very small miss rate at those input sizes. For larger problem sizes the differences are negligible. However, we see the number of retired instructions dropped by 16.3% and the number of retired NOPs dropped by 30%, a sign that issue bundles are filled with more useful instructions. We observed also an increase in memory parallelism in the transformed program, which lowers the exposed memory penalty and could be a factor for the increased performance.

Although we do not show any data, similar recurrences are in the I-line recursion loop with flux fixup at lines [416–469]. Which version of the loop is executed is controlled from the input file. The transformed code improves the execution time of that loop by a similar factor, and our measurements on Itanium2 confirmed this result.

We should mention that the performance increase obtained on Itanium2 might not be observed on every architecture. The I-line recursion loop contains a floating point divide operation (see Figure 6). If the throughput of the divider unit is low, or if the machine issue width is much lower, combined with possibly increased contention on other units, then the loop might be resource limited even in the original form, or the improvement could be only modest. However, our tool will predict correctly the lack of potential gains if the machine model is accurate. Itanium2 has a large issue width and floating point division is executed in software with a sequence of fully-pipelined instructions. Thus, while the latency of one divide operation from start to finish may be longer than what could be obtained in hardware, this approach is efficient when many divide operations need to be executed.

4.2 Analysis of POP

POP is a more complex application than Sweep3D and it has no single routine that accounts for a significant percentage of running time. Table 1 presents performance data for the top eight routines based on the potential for improvement from additional ILP. That data is predicted for an execution of POP 2.0.1 with the default benchmark size input on an Itanium2 machine model. In addition to the $MaxGain_{ILP}$ metric, Table 1 includes the predicted computation time and the rank of each routine if data was sorted by computation cost.

Unlike Sweep3D where effectively all execution time is spent in a single loop nest, in POP the execution cost is spread over a large number of routines. A traditional analysis looking for the most time consuming loops would not work well on this code since there is not a single loop or routine that accounts for a significant percentage of the running time. Sorting scopes by the $MaxGain_{ILP}$ metric enables us to at least limit our investigation to those scopes that show a non-negligible potential for improvement. The routine with the highest predicted potential for improvement, `boundary_2d_dbl`, at closer inspection proved to

[3] This metric does not include memory penalty.

[4] We tried -O3 as well, but -O2 yielded higher performance.

Routine	$\text{MaxGain}_{\text{ILP}}$		CpuTime		Rnk
boundary_2d_dbl	4.02e08	13.8%	6.99e08	6.2%	4
impvmixt	3.53e08	12.1%	6.17e08	5.5%	5
impvmixt_correct	3.45e08	11.8%	5.85e08	5.2%	7
global_sum_dbl	2.44e08	8.4%	3.38e08	3.0%	15
diag_global_preup	1.51e08	5.2%	3.26e08	2.9%	17
impvmixu	1.51e08	5.2%	3.26e08	2.9%	18
advt_centered	1.50e08	5.1%	3.85e08	3.4%	12
tracer_update	1.43e08	4.9%	7.78e08	6.9%	2

Table 1. POP: the top eight routines based on the improvement potential from extra ILP.

contain some frequently executed short loops which do not contain recurrences. The potential for improvement shown for this routine is the result of frequently executed loop exit paths that are not software pipelined. However, the following two routines, `impvmixt` and `impvmixt_correct`, contain loops with recurrences that account for most of the predicted improvement potential of these routines. These loops perform a tridiagonal solve in the vertical for every horizontal grid point and each tracer. Both these routines perform a very similar computation, thus what we describe below applies to both of them.

By visually inspecting the code of routine `impvmixt`, we realized that neither of the two outer loops carry any dependences, the computation for each horizontal grid point being independent. This means we can apply unroll & jam again to increase the ILP. However, the upper bound of the tridiagonal solve loop is a function of the ocean depth of each grid point. If we want to perform the tridiagonal solve for two grid points at a time, we must either ensure that they have the same depth, or compute the solve in parallel up to the minimum depth of a pair of points followed by a reminder loop that computes the residual points for the deepest point. Because the maximum depth specified in the benchmark input file is relatively small and the depths are integer values, we decided to implement the first solution. For this, we use a temporary array of size equal to the maximum possible depth to store the coordinates of the most recently encountered horizontal grid point with a given depth.

Routines `impvmixt` and `impvmixt_correct` are in fact very similar. They have almost identical code, including the tridiagonal solve loop. We transformed the code for routine `impvmixt_correct` using the same approach we used for the `impvmixt` routine. The other routines shown in Table 1 have a smaller potential for improvement from additional ILP. We found that with the exception of routine `impvmixu` which is similar to the routines presented above, the other routines do not contain real recurrences, but the potential for improvement is due to reduction operations that are not sufficiently parallelized by the Sun compiler used to compile the analyzed binary. We expect the Intel compiler will compute the reductions more efficiently given the fact it is targeted to a wider-issue architecture. We decided not to transform the code corresponding to routine `impvmixu` since this routine contributes less than 3% to the total computation time, and this percentage is even smaller once the memory penalty is considered.

We measured the performance of the original and the transformed versions of the code on our Itanium2 based machine. Since the two routines that were modified together account for only around 10% of the computation time in the original code, and since our transformations can in the best case scenario cut their computation cost in half, we expect at most a 5% improvement for the overall program. Once we consider the memory hierarchy delays that our transformations do not attempt to improve, the overall improvement that can be achieved should be even less. Using the default benchmark size input file, 192×128 horizontal grid points, we measured an overall performance increase of 3.78% for the transformed version. Increasing the number of grid points slightly to 208×144, the observed performance increase was 4.55%. While the overall improvement for this code is not substantial for reasons explained earlier, the transformation of the code was straightforward, and our performance modeling tool led us to these loops and predicted accurately the potential for improvement.

5 Related Work

Hardware counters provide a low overhead mechanism for monitoring the interactions between an application and its execution platform. A variety of tools use hardware performance counters to characterize the dynamic behavior of applications, e.g. HPCToolkit [12] and OProfile [13]. While these tools correlate resource utilization and architectural events with source programs, they don't provide insight into the factors that cause those events or into how performance would change if the machine characteristics were adjusted.

Many research groups use simulation or instruction schedulers to estimate application performance on a target architecture. Trimaran [5] provides an infrastructure for investigating the interplay between architecture parameters, compiler technology and applications. It evaluates overall application execution time using simulation. SLOPE [6] provides compiler based sensitivity analysis and performance prediction. It classifies memory references as strided or random and uses a simple list scheduler to compute instruction schedules for basic blocks, as the basis for static performance predictions. MonteSim [14], is a Monte Carlo simulator for predicting application performance on in-order microarchitectures. The simulation predicts the rate at which an application's instructions execute on a modeled architecture and how much time it will spend stalled. In

contrast, our work uses both static and dynamic analysis to provide application-centric performance feedback useful for tuning in addition to computing predictions of memory hierarchy and execution behavior.

6 Conclusions and Future Plans

This paper describes a performance modeling approach that can guide tuning by highlighting the factors that limit performance at points in a program. We describe two new performance metrics that can pinpoint sections of code with the highest potential for improvement from increased instruction level parallelism or from additional machine resources. These metrics can be directly computed by a modulo-scheduler. We presented a machine description language (MDL) that can model the main architecture characteristics affecting instruction scheduling and instruction latencies, and a modulo-scheduler targeted to such a machine model. The MDL's replacement rules are essential for producing accurate cross-architecture predictions. Applying our approach to the ASCI Sweep3D benchmark using an Itanium2 machine model, we found a loop with high potential for improvement from additional ILP. Transforming the loop increased overall application performance by 16% on an Itanium2-based platform. Applying our approach to POP, we identified routines with poor ILP and correctly predicted the potential for improving them. Transforming the code to increase ILP yielded 4% better performance.

This paper explores only one of the possible applications of these metrics. We are currently studying how to apply these and other metrics to understand which sections of a code could benefit from acceleration using an FPGA. Our future plans call for exploring how to extend our reuse distance based analysis of memory access patterns to identify how to improve memory hierarchy performance by applying loop transformations such as fusion and tiling.

7 Acknowledgments

This work is supported in part by the National Science Foundation under Cooperative Agreement No. CCR-0331654, the Department of Energy (DOE) Office of Science Cooperative Agreement No. DE-FC02-06ER25762, and DOE under Contract Nos. 03891-001-99-4G, 74837-001-03 49, 86192-001-04 49, and/or 12783-001-05 49 from the Los Alamos National Laboratory. Experiments were performed on equipment purchased with support from Intel and the National Science Foundation under Grant No. EIA-0216467.

References

[1] The ASCI Sweep3D Benchmark Code. DOE Accelerated Strategic Computing Initiative. `http://www.llnl.gov/asci_benchmarks/asci/limited/sweep3d/asci_sweep3d.html`.

[2] The Parallel Ocean Program (POP). `http://climate.lanl.gov/Models/POP`.

[3] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[4] B. Bennett and V. Kruskal. LRU Stack Processing. *IBM Journal of Research and Development*, 19(4):353–357, July 1975.

[5] L. N. Chakrapani, J. Gyllenhaal, W. W. Hwu, S. A. Mahlke, K. V. Palem, and R. M. Rabbah. Trimaran: An Infrastructure for Research in Instruction-Level Parallelism. *Lecture Notes in Computer Science*, 3602:32–41, 2005.

[6] P. C. Diniz and J. Abramson. SLOPE: A Compiler Approach to Performance Prediction and Performance Sensitivity Analysis for Scientific Codes. *Cyberinfrastructure Technology Watch: Special Issue on HPC Productivity*, 2(4B), Nov. 2006.

[7] R. A. Huff. Lifetime-Sensitive Modulo Scheduling. In *PLDI '93: Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, pages 258–267, New York, NY, USA, 1993. ACM Press.

[8] Intel Corporation. *Intel Itanium2 Processor Reference Manual for Software Development and Optimizations*, 2003.

[9] G. Marin and J. Mellor-Crummey. Cross-Architecture Performance Predictions for Scientific Applications Using Parameterized Models. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, pages 2–13. ACM Press, June 2004.

[10] G. Marin and J. Mellor-Crummey. Scalable Cross-Architecture Predictions of Memory Hierarchy Response for Scientific Applications. In *Proceedings of the Los Alamos Computer Science Institute Sixth Annual Symposium*, Santa Fe, NM, USA, Oct. 2005. `http://lacsi.rice.edu/symposium/symposiumdownloads/lacsi_2005/papers/pap108.pdf`.

[11] J. Mellor-Crummey. Using hpcviewer to Browse Performance Databases, Feb. 2004. `http://www.hipersoft.rice.edu/hpctoolkit/tutorials/Using-hpcviewer.pdf`.

[12] J. Mellor-Crummey, R. J. Fowler, G. Marin, and N. Tallent. HPCVIEW: A Tool for Top-down Analysis of Node Performance. *The Journal of Supercomputing*, 23(1):81–104, 2002.

[13] The OProfile website. `http://oprofile.sourceforge.net/docs`.

[14] R. Srinivasan and O. Lubeck. MonteSim: A Monte Carlo Performance Model for In-order Microachitectures. *SIGARCH Computer Architecture News*, 33(5):75–80, 2005.

[15] R. E. Tarjan. Testing Flow Graph Reducibility. *Journal of Computer and System Sciences*, 9:355–365, 1974.