Toward a Systematic, Pragmatic and Architecture-Aware Program Optimization Process for Complex Processors

David Parello, *ALCHEMY Group, HP France and HiPEAC network,*
Olivier Temam, *ALCHEMY Group, INRIA Futurs and LRI, Paris Sud University and HiPEAC network*,
Albert Cohen, *ALCHEMY Group, INRIA Futurs and HiPEAC network*,
Jean-Marie Verdun, *HP France*

ABSTRACT

Because processor architectures are increasingly complex, it is increasingly difficult to embed accurate machine models within compilers. As a result, compiler efficiency tends to decrease. Currently, the trend is on top-down approaches: static compilers are progressively augmented with information from the architecture as in profile-based, iterative or dynamic compilation techniques. However, for the moment, fairly elementary architectural information is used. In this article, we adopt a bottom-up approach to the architecture complexity issue: we assume we know everything about the behavior of the program on the architecture. We present a manual but systematic process for optimizing a program on a complex processor architecture using extensive dynamic analysis, and we find that a small set of run-time information is sufficient to drive an efficient process. We have experimentally observed on an Alpha 21264 that this approach can yield significant performance improvement on Spec benchmarks, beyond peak Spec. We are currently using this approach for optimizing customer applications.

1. INTRODUCTION

Because processor architectures are increasingly complex, it is increasingly difficult to embed accurate machine models within compilers. As a result, compiler efficiency tends to decrease as well as the rate of improvement of processor *sustained* performance. Currently, the trend is on top-down approaches: static compilers are progressively augmented with information from the architecture as in profile-based, iterative or dynamic compilation techniques. However, for the moment, only fairly elementary architectural information is used (most often execution time) in such feedback-directed compilation techniques. In this article, we adopt a bottom-up (and frontal) approach to the architecture complexity issue: assuming we know everything about the behavior of the program on the architecture (extensive dynamic analysis), what can we do to improve its performance? Based on extensive analysis of programs behaviors on a complex processor architecture, we have designed a systematic and iterative optimization process. While it is not yet implemented as a fully automatic iterative environment, it is systematic, and it has already been (and is still being) used successfully at HP France for the task of quickly optimizing programs on complex processors for prospective customers (on the Alpha for the moment; extension to x86 is planned).

Most current research works on iterative, feedback-directed optimization [4, 2] aim at proving the potential of iterative optimization itself. The goal of our research (and the resulting optimization process) is to address some of the *practical* issues that hinder the effective application of iterative optimization. Feedback-directed techniques [8, 4, 2] are currently limited to finding appropriate program transformation parameters, such as tile size, unroll factor, padding size, rather than the program transformation themselves, let alone compositions of program transformations; however, several recent research works have outlined that complex and variable compositions of program transformations can be necessary to reach high performance [19, 12], beyond the rigid sequence of program transformations embedded in static compilers. How can we find a proper composition of program transformations within such a huge search space? Currently, searching is restricted to a few optimizations, and even then, it usually requires several hundreds of runs using genetic algorithms or other operations research heuristics [8, 2, 16].

Our approach takes the form of a decision tree which guides the optimization process. Each branch of the tree is a sequence of analysis/decision steps based on run-time metrics (dynamic analysis), called *performance anomalies*, and a branch leaf is one or a few localized program transformation suggestions. An iteration of the optimization process is equivalent to walking down one branch. After the corresponding optimization has been applied, the program is run again, new statistics are gathered, the process starts again at the tree top and a new branch is followed. Progressively, the process builds a sequence (composition) of program transformations. The process repeats until further transformations do not bring any significant additional improvement. Granted, the process is just one of the many possible "walks" within a huge search space, but this walk is systematic; to a limited extent it provides an approach for whole-program optimization and it has been experimentally proved to yield significant performance improvement on Spec benchmarks [15], beyond peak Spec[1] and on customers applications. Table 1 summarizes the speedups achieved on the Spec using this process against different compiler optimizations and peak Spec; these experiments were run on an HP Alpha 21264 using the C (V6.4) and FORTRAN (V5.4) HP compilers.

Beyond a potential strategy for driving automatic iterative optimization environments in the future, this applied research work has several immediate benefits. (1) It provides a manual optimization process that can be used by engineers; because this process is systematic, less expertise is required on the part of the engineer to optimize a program. (2) The decision tree formalizes the empirical expertise of engineers, and it is a way to pass this expertise, traditionally hard to teach, to new engineers or researchers. (3) Each branch actually defines a mapping between a given architecture performance issue and appropriate program transformations; this mapping is based on empirical expertise. (4) Beyond the optimization

[1]Peak Spec performance is obtained with the best known combination of compiler flags, including profiling flags.

0-7695-2153-3/04 $20.00 (c)2004 IEEE.

	Peak Spec	Optimization Process	*Relative Speedup*
wupwise	1.20	2.90	*2.42*
equake	2.01	2.50	*1.24*
applu	1.47	2.18	*1.48*
swim	1.00	1.51	*1.51*
mgrid	1.59	1.45	*0.91*
facerec	1.04	1.42	*1.36*
ammp	1.18	1.40	*1.19*
galgel	1.04	1.39	*1.34*
apsi	1.07	1.23	*1.15*
mesa	1.12	1.17	*1.04*
fma3d	1.32	1.09	*0.82*
art	1.22	1.07	*0.87*

Table 1: *Speedup for 12 SpecFP benchmarks with respect to* Base Spec.

process, this empirical work also had the benefit of filtering which, among the many existing program transformations, bring the best benefits in practice.

In Section 2 we present our approach to dynamic analysis, in Section 3 how this analysis is used within the optimization process, and in Section 4 the experimental results on the SpecFP2000 benchmarks. In Section 5, we discuss the issues to be addressed for the automatization of the process.

2. PERFORMANCE ANOMALIES

The process relies on the observation of 14 different performance *anomalies*; some of these anomalies correspond to traditional statistics, e.g., D-TLB misses, available from program counters, and others are slightly more elaborate performance indicators. They aim at enumerating and separating the different possible causes of performance loss. Why do we need such "performance anomalies" and what are they exactly?

The initial motivation was to *find the exact cause of any performance loss* during a program execution, in order to apply the appropriate program optimization. In an out-of-order superscalar processor like the Alpha 21264, a performance loss occurs when, at a given cycle, the maximum number of instructions cannot be committed (11 in this case). Determining the cause of the performance loss means understanding why a given (or several) instruction(s) could not be committed. Determining the "real" cause for an instruction stall can be a very difficult task in such a processor because performance effects can propagate over a large number of cycles [3]: a data cache miss can slow down an arithmetic operation, which in turn has a resource conflict with another arithmetic operation, which in turn delays an address computation,... So that the instruction at the source of the performance loss may have left the pipeline many cycles before, and there are often multiple intertwined causes.

As a result, we have opted for a local, straightforward and brutal approach: we monitor key hardware components and say a performance loss *may* occur as soon as a hardware component is not performing *at full capacity*. More precisely, to characterize the performance loss induced by a given instruction, we restrict backtracking to the parent instructions only in the data-flow graph. And the selected program transformations will target that particular performance loss whether it is only a symptom or a cause. For instance in Figure 1: the analysis would naturally start with instruction *S*6 at the bottom of the data-flow tree, and it would be limited to its parents *S*4 and *S*5 in the grey areas.

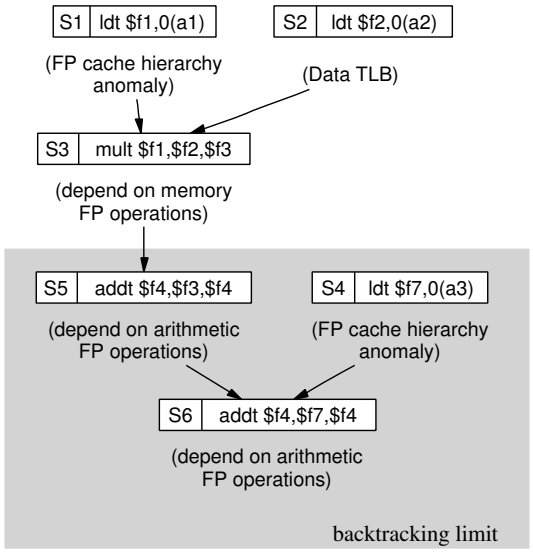

Figure 1: *Limited data-flow analysis.*

Characterizing the nature of the performance loss. To select the appropriate program transformation, we then attempt to better characterize the *nature* of performance loss (without further backtracking). The starting point of the analysis is again straightforward: a processor is performing at full capacity if all its functional units are used every cycle. So characterizing a performance loss means understanding why a functional unit is not busy. There are 4 main possible causes: a memory-bound program, a compute-bound program, lack of parallelism and architecture specific issues. We then try to precisely identify the nature of the performance anomaly. Assuming a functional unit is not used at full capacity, we examine the issue queue, determine whether there are instructions in it, and if so, we examine whether the operands are ready, and what are the source instructions, i.e., backtracking to the parents in the data-flow graph only. The status of the issue queue, the nature of the operands or architecture-specific metrics ("TRAPS" for the Alpha) will help precisely identify the nature of performance anomalies. The reasoning is pictured in the decision tree of Figure 2; how this tree was built is explained later on.

For instance, consider the branch highlighted with bold arrows in Figure 2. One performance anomaly indicates the number of cycles where a functional unit is starving with instructions in the issue queue but their operands are not ready. If that performance anomaly is dominant, then the program either lacks parallelism or is memory-bound. To distinguish between these two cases, we refine that metric by examining the parent instructions and counting the number of times the parent instruction is a load instruction or an ALU instruction. If it happens often that FUs are starving on a load instruction then the program is likely to be memory-bound; if FUs are starving more often on ALU operations, then the issue is lack of parallelism. Assuming, it is memory-bound, we then attempt to decide whether it is a memory *latency* or a memory *bandwidth* issue. For that purpose, we combine the previous performance anomaly with the average memory latency: if the average memory latency is low then it is a memory bandwidth issue, otherwise it is a memory latency issue, and so on. Naturally this decomposition is designed to narrow in on the most appropriate program transformation(s).

To collect such statistics, we examine individual functional units

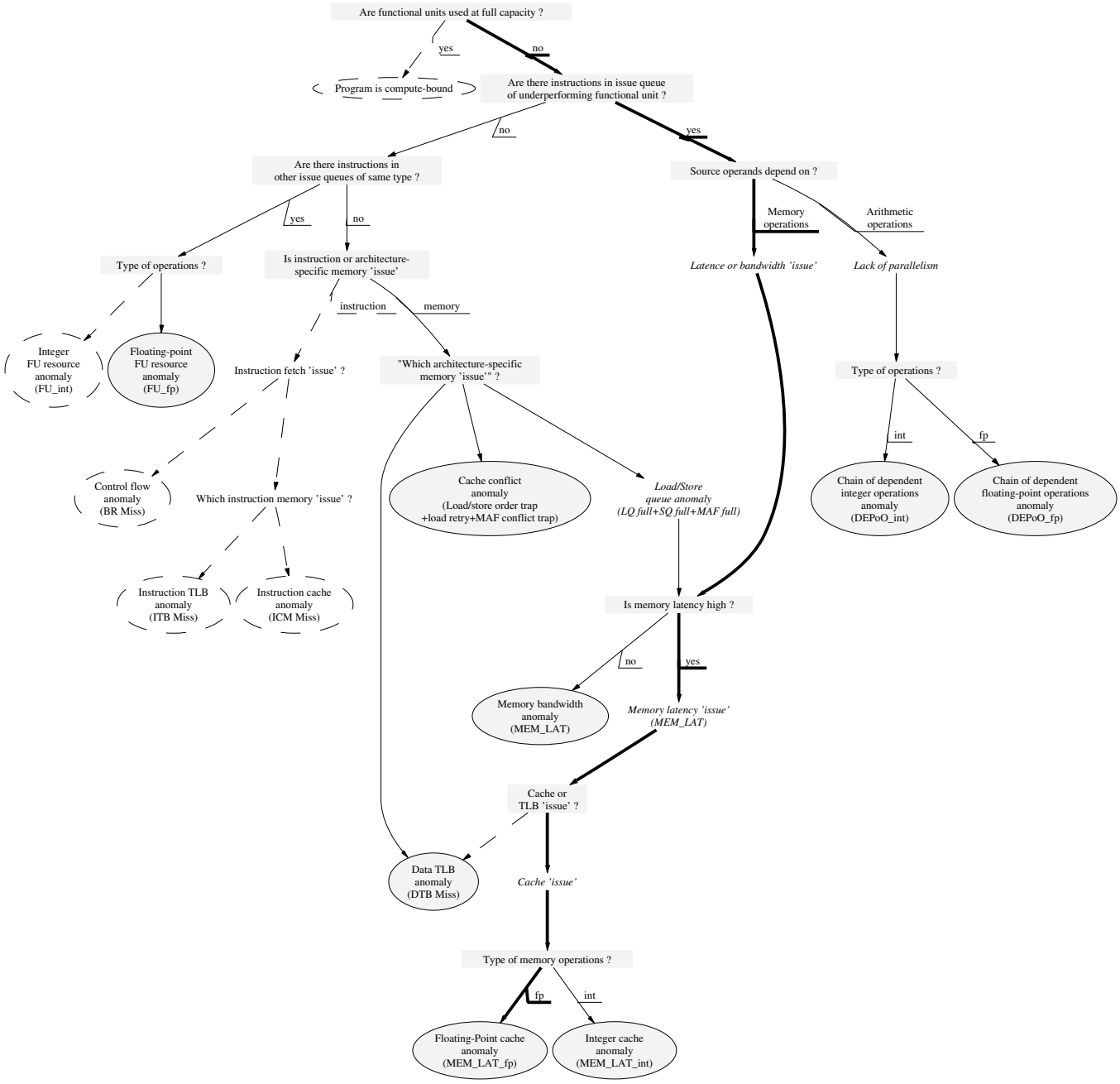

Figure 2: *Decision tree.*

at run-time every cycle. Consider again the example of Figure 1, and assume it illustrates the above case at a given cycle. The adder is starving with instruction $S6$ in the issue queue, its operand $\$f4$ is ready, and its operand $\$f7$ is not ready. The source instruction $S4$ of operand $\$f7$ is a load instruction, so we will increment a statistic counting dependencies on memory instructions (if the source instruction has completed, no statistic is incremented). Simultaneously, and independently, in accordance with the local analysis principle, we will record the latency of this load instruction and we will later find the latency is fairly small. If both statistics prove to be dominant, we will determine that instructions are usually waiting on memory instructions with low latencies and we will conclude the performance anomaly is a memory bandwidth issue. We found that, in many cases, such local analysis provides a sufficiently accurate characterization to drive optimization decisions.

Decision tree. To build the decision tree, we have initially gathered a large amount of empirical data: using a cycle-accurate Alpha 21264 processor simulator to provide detailed information on the program behavior on the architecture, we have optimized *by hand* 12 SpecFP benchmarks for this target processor over a period of 12 months, applying a large array of traditional loop transforma-

tion optimizations. For the first benchmark *applu*, the optimization effort was not structured except for the goal of achieving the best possible speedup. For the best optimization sequences, we have reverse-engineered the analysis and how the dynamic information was used to identify/characterize the nature of the performance loss. Then we have started to express the analysis process as a decision tree and we used the optimization of the remaining 12 benchmarks to either fine-tune the tree structure, add additional branches, and especially to map optimizations to performance anomalies, see Section 3. Figure 2 shows the corresponding decision tree. The difficulty is to strike the right *balance* between too little or too much information to drive the analysis process. Too little information and the characterization is not precise enough to indicate which optimizations to apply at the bottom of each branch; too much information and the decision tree is just a set of specific cases that do not generalize to other codes (or even code sections). Some of the branches in Figure 2 are dotted because the corresponding cases did not actually occur in SpecFP (such as instruction fetch issues), and as a result, the corresponding subtrees have not been explored.

Figure 2 also shows the different statistics/metrics collected to characterize anomalies and Table 2 defines these different statistics. Because the underlying philosophy of the approach was "assuming we know everything about the behavior of the program on the architecture…", we collected any statistics we felt could be necessary, we did not want to be limited by methodology issues. So the simulator was modified to insert the software probes required to collect some of the performance anomalies (especially, functional units anomalies). Naturally, simulator-based analysis is rather slow (even though recent techniques [5, 13, 18] can considerably speedup simulator-based program analysis), so in the next section we discuss how to collect such performance anomalies more efficiently.

Normalizing anomalies. To select the appropriate program transformations, we need to determine which performance anomaly is dominant. For that purpose, we must *normalize* the anomalies metrics. The first normalization is trivial and consists in expressing metrics as ratios: we simply divide most event counts by the total number of cycles in the program, i.e., number of D-TLB misses becomes number of D-TLB misses per cycle. Then, the issue is how to compare these different metrics? In other terms, we need to understand when a given metric value is "high" or "low". For that purpose, we have collected the anomalies metrics over all SpecFP2000 programs and we define a normalized value v_a^{norm} of a metric value v_a, for an anomaly a, as follows:

$$v_a^{norm} = \frac{v - m_a}{\sigma_a}$$

where m_a and σ_a are respectively the average and standard deviation for this metric over all SpecFP2000 (metrics are collected using the Base Spec versions). As a result, the normalized mean for any anomaly over all SpecFP2000 is is equal to 0, and the normalized standard deviation is equal to 1. In other terms, we use the SpecFP2000 as reference points for metric values. In the future, we will have to refine these reference points, for instance per application domain. Normalization is used only at the tree root to compare several unrelated metrics, e.g., lack of parallelism and memory latency. To compare related sub-metrics, e.g., lack of parallelism due to arithmetic or memory instructions, we use unmodified ratios; as a result, Table 2 only shows SpecFP2000 mean and standard deviation for root statistics.

Anomalies. Table 2 lists the hierarchy of anomalies as it is used in the optimization process. The four main categories of anomalies are: lack of parallelism, functional units anomalies, average memory latencies (it is mostly used to distinguish between memory *latency* and memory *bandwidth* issues), and partial pipeline flushes (either architecture-specific issues like replay traps for load/store conflicts upon cache accesses, or more general issues like load/store queue aliasing and miss address file overflow).

Collecting performance anomalies efficiently. For the moment, all anomaly statistics have been collected using a cycle-level simulator. We have investigated the possibility to collect some or all such statistics using existing performance counters on the Alpha 21264 and on a recent processor, the Itanium2.

Alpha processors have a limited support for hardware monitoring. Unfortunately, this limitation is most stringent on the 21264, which exposes only 9 events. Using the OProfile toolkit [10], one may yet retrieve a subset of the anomaly statistics.

PerfMon3 [14] is the programmer interface and performance-monitoring tool to the hundreds of hardware counters of the Itanium2 [6]. It simultaneously monitors 4 independent event counters, with a negligible overhead to start/stop monitoring at given program counter addresses or program symbols. Event reporting can be narrowed to a specific instruction address range or opcode (for most instruction-attached events) and to a data address range (for memory accesses).

Both monitoring tools have a good support for event sampling, to report statistics at a customizable rate, hence to localize the anomalies. Interestingly, monitoring for the Alpha or Itanium2 processors does not induce any code instrumentation; but fine-grain narrowing of the measurements, e.g., to a few tenths of instructions, may require the insertion of explicit start/stop instructions.

We evaluated the compatibility of the Alpha and Itanium2 hardware counters with our anomaly reporting scheme. Table 3 maps the statistics of Table 2 to the corresponding hardware counters, whose semantics should be intuitive from their names (more details for the Itanium2 can be found in [6]). Blank cells denote anomaly statistics that cannot be monitored with hardware counters. Asterisks * denote modified statistics to match the in-order pipeline and decoupled front-end of the Itanium2, where counting bubbles makes more sense than counting misses.

The Alpha 21264 hardware counters are coarser. Few anomaly statistics can be evaluated, and the synthetic `DEP` anomaly may be roughly estimated from the number of retired instructions, the cycle count, and the number of pipeline flushes (mostly traps and branch mispredicts). Conversely, if we were to extend our decision tree to the Itanium2, its hardware counters would come close to satisfying all our monitoring needs. They do unfortunately not differentiate the arithmetic and memory causes of starving floating-point units: we may only compute a synthetic `DEP_fp` statistic. This limitation also makes the computation of `MEM_LAT_fp` more challenging: thanks to the Itanium2 multi-occurrence reporting scheme, it is possible to track one pending floating-point load (randomly chosen) at a time and report its latency; the average floating point load latency can be derived from the accumulated latency by statistical extrapolation to the non-reported loads (thanks to the `DATA_EAR_EVENTS` counter). These statistical monitoring options are available in Intel's VTune performance analysis tool [17]. Notice the `Cache-conflict` anomaly is specific to the Alpha. Conversely, specific anomalies could be analyzed to refine the decision tree for the Itanium2: for example, the closest to the `Cache conflict` anomaly would be to the `L2_OZQ_CANCELS1.BANK_CONF` event, i.e., conflicts between the 16 banks of the L2 cache, etc.

Statistics	Description	Mean value (SpecFp2000)	Standard deviation (SpecFp2000)
DEP	**Lack of parallelism (DEP_fp + DEP_int)**	**7.38**	**5.08**
DEP_fp	DEPoO_fp + DEPoM_fp		
DEPoO_fp	Number of times per cycle a functional unit is starving and an instruction in the issue queue is waiting for an operand produced by a floating-point arithmetic operation		
DEPoM_fp	Number of times per cycle a functional unit is starving and an instruction in the issue queue is waiting for an operand produced by a floating-point memory operation		
DEP_int	DEPoO_int + DEPoM_int		
DEPoO_int	Number of times per cycle a functional unit is starving and an instruction in the issue queue is waiting for an operand produced by an integer arithmetic operation		
DEPoM_int	Number of times per cycle a functional unit is starving and an instruction in the issue queue is waiting for an operand produced by an integer memory operation		
FU	**Functional units anomalies (FU_fp + FU_int)**	**2.09**	**0.54**
FU_fp	Number of times per cycle a floating-point functional unit is starving and no dedicated floating-point instruction is in issue queue		
FU_int	Number of times per cycle a integer functional unit is starving and no dedicated integer instruction is in issue queue		
MEM_LAT	**Average memory latency of all loads**	**12.84**	**8.13**
MEM_LAT_fp	Average memory latency of floating-point loads		
MEM_LAT_int	Average memory latency of integer loads		
Traps	**Architecture-specific anomalies (ITraps + DTraps)**	**0.005**	**0.041**
ITraps	BR Miss + IC Miss + ITB Miss		
BR Miss	Number of branch mispredictions per cycle		
IC Miss	Number of instruction cache misses per cycle		
ITB Miss	Number of ITLB misses per cycle		
DTraps	DTB Miss + Cache conflict + LSQ/MAF full		
DTB Miss	Number of DTLB misses per cycle		
Cache conflict	Number of cache access conflicts per cycle, due to a load and a store access to the same cache line		
LSQ/MAF full	Number of the number of times per cycle where the load queue, store queue or miss address file is full		

Table 2: *Statistics used to characterize anomalies.*

3. OPTIMIZATION PROCESS

The iterative optimization processes proposed up to now [8, 2] essentially rely on random searches to find the appropriate transformation parameters (or small sequences of transformations) for a few selected transformations. Assuming there is no restriction on the possible transformations, and that we want to find possibly long sequences of transformations, the search space becomes huge. Our approach is not designed to bring optimal performance but to ensure a steady performance progression across iterations. We have experimentally observed that it can yield significant performance improvements, which implicitly, either means there are many possible walks to increased performance or that a walk based on improving the local behavior of hardware components is efficient.

Our process is iterative, and each iteration is simply decomposed into the following steps:

1. The performance anomalies are collected.
2. The decision tree is used to narrow in on the performance issue.
3. The location (source statement or even assembly instruction) with the worst behavior for this particular issue is identified.
4. For each performance issue, a set of program transformations is suggested. The program transformations with the most local effects are preferred (e.g., loop tiling is preferred over loop merging), assuming there is no legality issue.
5. The program is run again and another iteration starts.

Implicit data-flow analysis through iterations. As explained in Section 2, in order to characterize the performance loss induced by a given instruction, we restrict the analysis to the parent instructions only in the data-flow graph. This *local* (and thus limited) analysis means that, in some cases, the selected program transformations will fight the most obvious cause for a performance loss but not several *nested* (combined) causes along the data-flow tree. The reason why such limited analysis is in fact sufficient is related to the *iterative* nature of the process: at any iteration, the process quantifies and normalizes the different anomalies in order to determine the most critical one. Then it optimizes the program with a transformation targeting this anomaly; then anomalies are evaluated again, and usually another anomaly becomes dominant. Progressively, the process has the effect of *hiding* the most obvious causes and unveiling new causes of performance loss. Implicitly, the data-flow analysis occurs *through* the process iterations.

Consider for instance *applu*. During the optimization process, we found a lack of parallelism anomaly (a chain of dependent FP instructions) localized in the loop nest of Figure 3(a) in procedure *blts*, and more particularly on the subtract operation (`v(m,i,j,k) - omega`). Because this operation depends on 3 multiplications and 3 additions, we found loop splitting to be effective in that case (one of the optimizations used to break chains of dependent operations); loop splitting is implemented by introducing a temporary array where the computation between parenthesis is stored

Statistics	Alpha counters	Itanium2 counters
DEP	*coarse approximation possible*	DEP_fp + DEP_int
DEP_fp		BE_EXE_BUBBLE.FRALL
DEPoO_fp		
DEPoM_fp		
DEP_int		DEPoO_int + DEPoM_int
DEPoO_int		BE_EXE_BUBBLE.GRGR
DEPoM_int		BE_EXE_BUBBLE.GRALL − BE_EXE_BUBBLE.GRGR
FU		FU_fp + FU_int
FU_fp		CPU_CYCLES / (FP_OPS_RETIRED + L2_OPS_ISSUED.FP_LOAD)
FU_int		CPU_CYCLES / (IA64_INST_RETIRED − FP_OPS_RETIRED − L2_OPS_ISSUED.FP_LOAD)
MEM_LAT		MEM_LAT_fp + MEM_LAT_int
MEM_LAT_fp		PMC_11 / DATA_EAR_EVENTS *PMC_11 is configured as a multi-occurrence latency counter*
MEM_LAT_int		BE_L1D_FPU_BUBBLE.L1D / L1D_READS
Traps	REPLAY_TRAP	ITraps + DTraps
ITraps		BR MISS + IC MISS + ITB MISS
BR Miss	MISPREDICTS / CYCLES	(BE_BR_MISSPRED_DETAIL.WRONG_PATH + BE_BR_MISSPRED_DETAIL.WRONG_TARGET) / CPU_CYCLES
IC Miss*		BE_LOST_BW_DUE_TO_FE.IMISS / CPU_CYCLES
ITB Miss*	ITB_MISS / CYCLES	BE_LOST_BW_DUE_TO_FE.TLBMISS / CPU_CYCLES
DTraps		DTB MISS + Cache conflict + LSQ/MAF full
DTB Miss	DTB_MISS / CYCLES	L2DTLB_MISSES / CPU_CYCLES
Cache conflict		*irrelevant on the Itanium2*
LSQ/MAF full		L2_FILLB_FULL + L2_OZDB_FULL + L2_OZQ_FULL

Table 3: *Hardware counters to collect statistics efficiently.*

(the 3 multiplications and 3 additions). At the next iteration, performance had improved and lack of parallelism was no longer a dominant performance anomaly; the new anomaly proved to be a memory latency issue localized in array references `ldx`, `ldy`, `ldz`. This problem was not created by loop splitting, it was only hidden, performance-wise, by the parallelism issue in the subtract operation. If we had backtracked through the data-flow graph, we might have detected both issues simultaneously, but we would have had to come up with a program transformation that addresses multiple performance anomalies simultaneously, which can be a significantly more complex optimization problem.

```
do k = 2, nz-1
  do j = 2, ny-1
    do i = 2, nx-1
      do m = 1, 5
        do l = 1, 5
          v(m,i,j,k) = v(m,i,j,k) - omega *
             (   ldz(m,l,i,j,k) * v(l,i,j,k-1)
               + ldy(m,l,i,j,k) * v(l,i,j-1,k)
               + ldx(m,l,i,j,k) * v(l,i-1,j,k)
```

(a) *blts* procedure

```
do k = nz-1, 2, -1
  do j = ny-1, 2, -1
    do i = nx-1, 2, -1
      do m = 1, 5
        tv(m) = 0.0d+00
        do l = 1, 5
          tv(m) = tv(m) - omega *
             (   udz(m,l,i,j,k) * v(l,i,j,k+1)
               + udy(m,l,i,j,k) * v(l,i,j+1,k)
               + udx(m,l,i,j,k) * v(l,i+1,j,k)
```

(b) *buts* procedure

Figure 3: *Similar code structures eligible for* generalization.

Enabling transformations. We found that program transformations can often benefit from or require the application of other program transformations. Consider for instance *swim*; we found an optimization sequence that includes loop merging, then padding and tiling. *swim* contains 3 large loop nests and loop merging replaces all 3 loop nests with a single one. Padding and tiling in *swim* bring less performance improvement if they are applied to the 3 separate loop nests of *swim* rather than to a single merged nest. However, loop merging itself cannot be applied before shifting is applied to the loop in procedure *calc3* because a dependency prevents loop merging. In summary, padding and tiling benefit performance-wise from the application of loop merging, and loop merging itself cannot occur before loop shifting is applied.

It is really the role of dynamic analysis to find program transformations that *benefit* others, i.e., *enabling* transformations in a performance sense, or simply put, the best performing sequence of program transformations. On the other hand, we believe that finding transformations that make it legal to apply other transformations (i.e., *enabling* transformations in the legality sense) is well suited and within reach of static analysis (unlike the performance analysis tasks currently devoted to static analysis). In our optimization process, the latter is currently done by hand; it is actually the only task that, for the moment, prevents the optimization process from being fully automatic, see Section 5. The former is implicitly implemented through *iteration backtracking*.

Backtracking iterations and allowing temporary performance loss. Indeed, in the first iterations of *swim*, the first performance anomalies were properly addressed using padding then loop tiling, and the execution time dropped from 204 seconds down to 170 seconds. Further performance anomalies (memory latency issues) could not be addressed until we *backtracked* the process and started with loop merging instead of padding and tiling. The sequence bringing the best performance improvements proved to be

loop merging then padding then tiling, because the memory latency issues were due to non-exploited inter-nest locality [9].

For that reason, the process always allows for 2 consecutive iterations without performance improvement, then it backtracks to the previous iteration, picks one of the other possible optimizations and moves forward again. When all suggested transformations have been tried, it backtracks further again to the previous iteration and so on.

Localization and generalization. Once the anomaly is identified, the source statements with the worst behavior for that anomaly are located. It often happens that several source statements, located in distant program sections, exhibit almost the same value for the dominant anomaly. The reason is that many program sections can have the same structure and thus exhibit the same behavior (and performance issue). In such cases, we have experimented with *generalizing* the program optimization, i.e., applying the same program optimization to all such program sections. Because all optimizations were performed by hand, we could not afford to systematically apply generalization, it was only experimented in a few cases.

Consider for instance the optimization of benchmark *applu*. At iteration 1, we found the dominant anomaly is *Lack of parallelism*. The localization steps indicated that 2 source statements had a similar performance issue, corresponding to different code sections. We focused on the first statement at line 646 in procedure *blts*, see Figure 3(a), and solved the performance issue with loop fusion. Examining the other source statements reveals a very similar structure at line 772, corresponding to the loop nest in procedure *buts*, see Figure 3(b). Running again the analysis might show the performance anomaly is no longer dominant because the main source statement has been dealt with, but potential performance benefits have been lost because the performance anomalies at the other source statements remains. Therefore, in the cases where the program structure is almost identical in the different target source statements, we have experimented a few times with *generalizing* the optimization, i.e., simply applying the same optimization to the next source statements. Usually, we first run another analysis step after the application of the first source statement to validate its effect on the target performance anomaly, then we generalize the optimizations, and then look again for the new dominant performance anomaly. In the case of *wupwise*, applying the optimization a single time decreased execution time from 232 seconds to 135 seconds; generalizing the optimization further decreased the execution time to 80 seconds.

Table 4 shows the contribution to speedup (in percentage) of the generalization steps for some of the benchmarks. While the process can be implemented without generalization, it improves its efficiency.

Contribution of generalization	
equake	9%
wupwise	41%
applu	38%
mgrid	6%
galgel	8%
Average	20%

Table 4: *Contribution of* generalization *to the final speedup.*

Mapping performance issues to program transformations. As explained above, the optimization process was built empirically in a trial and error way, especially for the first benchmarks. We have tested many iterations per benchmark with the purpose of finding the appropriate mapping between a given performance issue and the program transformations that perform best *in practice* for this performance issue. Table 5 provides a mapping between the decision tree leafs and the suggested (filtered) program transformations. The table indicates the different cases where a given program transformation was found to bring a positive effect (at least 5% improvement on the targeted performance anomaly). Not all program transformations have the same impact; the table also outlines the average speedup brought by each transformation and number of times each program transformation was used, thereby providing a transformation ranking used to decide in which order program transformations must be tested (besides how local their effects are).

Optimizations	FP cache anomaly	INT cache anomaly	Memory bandwidth	Chain of dependent FP operations	Chain of dependent INT operations	FP FU resource anomaly	Cache conflict anomaly	Data TLB	Average speed-up	# Applications
Data-layout	√						√	√	1.34	3
Scheduling					√		√		1.24	2
Loop fission		√	√	√		√			1.21	4
Register blocking			√		√	√			1.14	5
Inlining		√			√	√			1.14	3
Unrolling					√	√			1.11	2
Padding	√								1.10	1
Tiling	√								1.10	1
Optimized libraries					√				1.10	1
Shifting	√		√	√	√				1.06	9
Scalar promotion					√		√	√	1.06	6
Loop merging			√	√		√	√	√	1.05	8
Loop interchange		√					√	√	1.02	5
Number of applications	3	2	8	6	7	5	8	5		

Table 5: *Mapping between anomalies and program transformations.*

Optimization process versus static compilers. The main asset of such an optimization process is increased flexibility compared to static compilers. Part of this flexibility comes from the capacity to compose long sequences of transformations, such as *galgel* in Figure 4 where the final optimization sequence is a composition of 7 program transformations. And part comes from the manual application of transformations which overrides the limitations of syntax-based intermediate representations.

First, a compiler usually has a fixed optimization strategy allowing little variation in the composition of transformations; for instance, in the ORC compiler [11], loop merging and loop fission can be interchanged and repeatedly applied, but it is not the case for many other transformations; compilers lack the flexibility to apply varying sequences of transformations. Second, program transformations are currently applied in a syntactic way, meaning a new syntactic (syntax-based) representation of the program is generated within the intermediate representation after each program transformation. Such syntactic representations are not compatible with long sequences of transformations, because one transformation can have an adverse effect on the application of further transformations, e.g., loop peeling, often used for merging two loops, can later hamper the application of loop tiling. For that purpose,

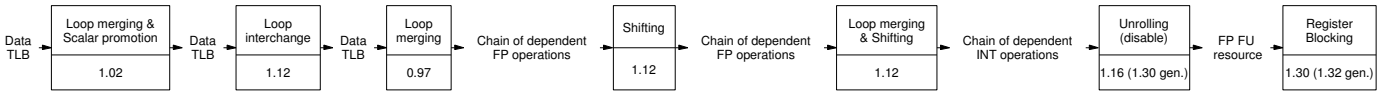

Figure 4: *Optimization sequence for galgel.*

we are investigating the utilization of elaborate program representations [1] compatible with the application of long sequences of complex program transformations.

4. EXPERIMENTAL RESULTS

The purpose of this section is to illustrate the process in detail with one SpecFP2000 benchmark and to give an overview of the experimental results obtained with the other SpecFP2000 benchmarks. We manually optimized 12 SpecFp2000 benchmarks (out of 14) and were able to outperform the peak Spec performance [15] (obtained in choosing the most appropriate compiler flags) for 9 of them, by stopping the optimization process after 4 iterations on average.

The experimental platform is an HP AlphaServer ES45, 1 GHz Alpha 21264C EV68 (1 processor enabled) with 8 MB L2 cache and 8 GB of memory. We compare our optimized versions with the *Base* Spec performance, i.e., the output of the HP Fortran (V5.4) and C (V6.4) compiler (`-arch ev6 -fast -O5 ONESTEP`) using the KAP Fortran preprocessor (V4.3). We also compare with the *Peak* Spec performance. Figure 1 summarizes the speed-up of peak Spec and of our manual optimizations, both with respect to the Base Spec performance, as well as the relative speedup (our optimizations versus Peak Spec).

From a program transformation point of view, our methodology results in a structured sequence of transformations applied to various code sections. In the examples below, for each program, we focus on 1 to 3 code sections where several transformations are iteratively applied. On average, we modified 30 source lines per program; the amount of modifications was essentially limited by the fact the process was manual.

Putting it all together. Let us consider the *wupwise* benchmark; its Base Spec execution time is 232 seconds.

Iteration 1. The global statistics of functional unit utilization show that functional unit `fadd` is not fully utilized, and that no instruction is available in its issue queue, indicating a lack of parallelism. The problem is then localized to procedure *gammul* and the statements with the highest anomaly ratings correspond to procedure calls *zcopy* and *zaxpy*, see Figure 5. Naturally, whenever localization targets a statement, or a code construct (e.g., loop nest), with procedure calls, they are inlined; inlining may be sufficient provided the compiler can then perform additional transformations, but it is not the case here, the performance remains unchanged.

```
...
CALL ZCOPY(12, X, 1, RESULT, 1)
CALL ZAXPY( 3,  I, X(10), 1, RESULT( 1), 1)
CALL ZAXPY( 3,  I, X( 7), 1, RESULT( 4), 1)
CALL ZAXPY( 3, -I, X( 4), 1, RESULT( 7), 1)
CALL ZAXPY( 3, -I, X( 1), 1, RESULT(10), 1)
...
```

Figure 5: *Procedure gammul from wupwise.*

Iteration 2. Now the same performance anomalies occur at the same source statements. Since the performance anomaly is the same (lack of parallelism), we test different program optimizations, checking which optimizations can be applied, starting with the transformations having the most local effects. In that case, we select loop unrolling; since the loop bounds values are available, full unrolling is performed. The code after inlining and full unrolling is shown in Figure 6. The execution time is now 146 seconds.

```
...
RESULT( 1) = X( 1) + (I * X(10))
RESULT( 2) = X( 2) + (I * X(11))
RESULT( 3) = X( 3) + (I * X(12))
RESULT( 4) = X( 4) + (I * X( 7))
...
```

Figure 6: *Procedure gammul from wupwise after inlining and full unrolling.*

```
...
CALL GAMMUL(1,0,X(1,(IP+1)/2,J,K,L),AUX1)
CALL SU3MUL(U(1,1,1,I,J,K,L),'N',AUX1,AUX3)
CALL GAMMUL(2,0,X(1,(I+1)/2,JP,K,L),AUX1)
CALL SU3MUL(U(1,1,2,I,J,K,L),'N',AUX1,AUX2)
CALL ZAXPY(12,ONE,AUX2,1,AUX3,1)
CALL GAMMUL(3,0,X(1,(I+1)/2,J,KP,L),AUX1)
CALL SU3MUL(U(1,1,3,I,J,K,L),'N',AUX1,AUX2)
CALL ZAXPY(12,ONE,AUX2,1,AUX3,1)
CALL GAMMUL(4,0,X(1,(I+1)/2,J,K,LP),AUX1)
CALL SU3MUL(U(1,1,4,I,J,K,L),'N',AUX1,AUX2)
CALL ZAXPY(12,ONE,AUX2,1,AUX3,1)

CALL ZCOPY(12,AUX3,1,RESULT(1,(I+1)/2,J,K,L),1)
...
```

Figure 7: *Procedure muldeo from wupwise.*

Iteration 3. The same anomalies occur, but located on enclosing procedure calls, in the *muldeo* procedure, see Figure 7. Once again, we perform inlining and full unrolling of the inner loops. The resulting code is very large, all procedure calls (with their inner loops) in *muldeo* have been converted to straight-line code. Fig-

```
...
AUXP1( 1) = X( 1,(IP+1)/2,J,K,L)
          + (IMA * X(10,(IP+1)/2,J,K,L))
AUXP1( 2) = X( 2,(IP+1)/2,J,K,L)
          + (IMA * X(11,(IP+1)/2,J,K,L))
...
AUXP1(12) = X(12,(IP+1)/2,J,K,L)
          + ((-IMA) * X( 3,(IP+1)/2,J,K,L))

AUXP3( 1) = U(1,1,1,I,J,K,L) * AUXP1( 1)
          + U(1,2,1,I,J,K,L) * AUXP1( 2)
          + U(1,3,1,I,J,K,L) * AUXP1( 3)
...
AUXP3(12) = U(3,1,1,I,J,K,L) * AUXP1(10)
          + U(3,2,1,I,J,K,L) * AUXP1(11)
          + U(3,3,1,I,J,K,L) * AUXP1(12)

RESULT( 1,(I+1)/2,J,K,L) = AUXP3( 1)
RESULT( 2,(I+1)/2,J,K,L) = AUXP3( 2)
...
```

Figure 8: *Procedure muldeo from wupwise after inlining and full unrolling.*

ure 8 shows fragments of this large block.

Iteration 4. Once again, the same anomalies remain: lack of parallelism at the inlined source statements. Since there is no more loops to unroll, we attempt to extract more parallelism at the statement level using scalar promotion. The code after substitution of temporary arrays `AUXP*` is shown in Figure 9. Execution time is now 135 seconds.

```
AUXP1_1 = X( 1,(IP+1)/2,J,K,L)
      + (IMA * X(10,(IP+1)/2,J,K,L))
AUXP1_2 = X( 2,(IP+1)/2,J,K,L)
      + (IMA * X(11,(IP+1)/2,J,K,L))
    ...
AUXP1_12 = X(12,(IP+1)/2,J,K,L)
      + ((-IMA) * X( 3,(IP+1)/2,J,K,L))

AUXP3_1 = U(1,1,1,I,J,K,L) * AUXP1_1
      + U(1,2,1,I,J,K,L) * AUXP1_2
      + U(1,3,1,I,J,K,L) * AUXP1_3
RESULT( 1,(I+1)/2,J,K,L) = AUXP3_1
AUXP3_2 = ...
```

Figure 9: *Procedure muldeo from wupwise after scalar promotion.*

Generalization. A new run indicates the same anomaly in a different code section, with the same or similar procedures being called. We then decide to *generalize* the transformation to all statements calling these procedures and to perform the same transformation. The execution time now decreases to 80 seconds.

We have not pursued the optimization further for this program; if we had, the process would now focus on *cache conflict* anomalies.

Sequences of transformations. Table 6 summarizes the analyses/optimizations of the 9 SpecFP2000 benchmarks for which we could outperform the Peak Spec performance. Each row contains the sequence of transformations applied to one or several code sections, including generalization steps. Each bracket corresponds to a program transformation, with the acronym of the program transformation above, together with the corresponding speedup at the top (the speedup is always given with respect to the original program execution time), and the corresponding performance anomaly below. We stopped iterating whenever we reached significant speedup or after 3 iterations. An automatic process would have probably unveiled much more optimization opportunities.

As stated in Section 3, the variety and length of the transformation sequences explain why our approach outperforms current compilers. Indeed, most transformations in Table 5 are implemented in state-of-the-art compilers, but due to syntactic (pattern-matching), static analysis (dependences) and phase-ordering limitations, the Peak Spec switches failed to trigger important optimizations. *applu*, *swim* and *apsi* stress two additional difficulties for traditional compilers:

- optimizations on independent code fragments may follow opposite directions (for *applu*, loop merging and loop fission, array privatization and scalar promotion);
- optimizations with a global impact on the code structure may only be identified after the application of local transformations, forcing to backtrack and replay the whole sequence (loop merging for *swim*, *applu* and *apsi*).

Most efficient program transformations. Table 5 indicates how many times each program transformation had a positive effect (more than 5% improvement on the targeted performance anomaly), and the average speedup induced by the application of this program transformation. Beyond the optimization process, this empirical work also has the benefit of filtering out which, among many of the existing program transformations, bring the best benefits in practice. For instance, we found that far too much attention has been devoted to the memory *latency* issue, and far too little attention has been devoted to the memory *bandwidth* issue. As a result, largely investigated program transformations like tiling for caches bring fairly low benefits in most cases. On the other hand, less popular program transformations like forward substitution or tiling for registers can significantly improve memory bandwidth and overall performance. Several other architecture-related issues have been neglected as well, like locality across distant program regions, replay traps, interactions between memory components, functional unit usage,...

For example, *equake* presents several atypical cases of optimizations. In the first process iteration, a memory latency anomaly suggests the application of a data layout optimization which linearize as 3-dimensional array implemented using pointers; a second iteration reveals an architecture-specific anomaly, i.e., a load/store conflict due to a load immediately followed by a store at the same address (and no other statement in the loop), and the performance issue is resolved using loop merging which has the effect of allowing the compiler to increase the distance between a load and a store; overall the execution time decreases from a Base Spec of 290 seconds to 116 seconds.

Additional benefits of an iterative process. As part of a benchmarking activity, we have applied this optimization process to customers applications. In some cases, we found it particularly useful to be able to precisely control the optimization process. For instance, for one large customer application ($\sim 800,000$ lines), the different procedures are compiled using different flags and the optimization level for each procedure is reduced until the numerical stability of the application is observed. As a result, some code sections are poorly optimized and an iterative optimization process enables to monitor numerical stability while fine-tuning the application performance. Moreover, finding optimization opportunities is rather easy in that context; in one example, a lack of parallelism anomaly pointed to a reduction. We unrolled it by introducing four temporary scalar variables, and performed a final sum (reduction); we verified the operation did not affect numerical stability and achieved a 1.25 speedup thanks to this simple transformation. Note that this optimization can be automatically applied by a compiler, except for the numerical stability issue.

5. TOWARDS AN AUTOMATION OF THE PROCESS

We are already using the current process manually, progressively updating the analysis tree and especially the mapping between performance issues and program optimizations with each new program (scientific applications only). At the same time, we are also working on automating the process. This research is divided into two parts: (1) a faster method for collecting information with almost the same semantics as performance anomalies using hardware counters, as explained in Section 2, and (2) using static analysis to enumerate the different opportunities of application of the suggested optimizations. The latter part consists in replacing with static analysis the current manual code inspection to find how to apply a suggested optimization to the target source statement. The role of static analysis is really twofold: to find the *opportunities of application* of program transformations and to check legality issues; but the role of static analysis is *not* to decide where and especially *if* a pro-

Benchs	Optimization chains	Speedup
wupwise	[Opt. Library 1.10 / Chain of dep. INT] ▶ [INLINING/Unrolling 1.21 / FP FU resource] ▶ [INLINING/Unrolling 1.59 / FP FU resource] ▶ [Scalar Promotion/SCHEDULING 1.72 / Bandwidth] ▶ [General. 2.90]	2.90
swim	[Padding/Tiling. 1.20 / FP cache] ▶ [Loop MERGING 1.51 / Bandwidth] ▶ [1.51]	1.51
mgrid	[Loop FISSION 0.75 / FP FU resource] ▶ [Loop FISSION 1.27 / Bandwidth] ▶ [General. 1.32]	1.32
	▶ [Reg.BLOCKING 1.38 / Bandwidth] ▶ [General. 1.41]	1.41
applu	[Shifting 1.02 / Chain of dep. FP] ▶ [Scalar Promotion 1.04 / Chain of dep. FP] ▶ [General. 1.06]	1.06
	▶ [ShiftingBreak/Loop MERGING 1.08 / FP cache] ▶ [SCHEDULING 1.20 / Bandwidth] ▶ [General. 1.48]	1.48
	▶ [Loop FISSION 1.51 / Chain of dep. FP] ▶ [Unrolling (Full) 1.65 / FP FU resource] ▶ [General. 2.18]	2.18
mesa	[INLINING 1.06 / Chain of dep. INT] ▶ [Shifting 1.13 / Chain of dep. INT] ▶ [DataLayout 1.13 / Cache conflict] ▶ [Shifting 1.19 / Chain of dep. INT] ▶ [1.19]	1.19
galgel	[Loop MERGING/Scalar Promotion 1.02 / Data TLB] ▶ [Loop INTERCHANGE 1.12 / Data TLB] ▶ [Loop MERGING/Scalar promotion 0.97 / Data TLB]	
	▶ [Shifting 1.12 / Chain of dep. INT] ▶ [Loop MERGING/Shifting 1.12 / Chain of dep. FP] ▶ [Unrolling (Disabled) 1.16 / Chain of dep. INT] ▶ [General. 1.24]	1.24
	▶ [Reg.BLOCKING 1.30 / FP FU resource] ▶ [General. 1.32]	1.32
art	[Loop INTERCHANGE/Scalar Promotion 1.02 / INT cache] ▶ [Loop FISSION 1.07 / INT cache] ▶ [1.07]	1.07
equake	[DataLayout 1.70 / FP cache] ▶ [General. 1.92]	1.92
	▶ [Loop MERGING1.95 / Cache conflict] ▶ [Scalar Promotion. 2.50 / Cache conflict] ▶ [2.50]	2.50
facerec	[Reg.BLOCKING 1.30 / Chain of dep. FP] ▶ [1.30]	1.30
ammp	[Shifting 1.01 / Chain of dep. INT] ▶ [Shifting 1.02 / Chain of dep. INT] ▶ [Loop MERGING 1.06 / FP FU resource] ▶ [1.06]	1.06
fma3d	[Shifting 1.00 / Chain of dep. INT //] ▶ [Loop FISSION 1.00 / Cache conflict] ▶ [Loop MERGING 1.03 / Chain of dep. FP] ▶ [INLINING 1.09 / Bandwidth.] ▶ [1.09]	1.09
apsi	[DataLayout 1.07 / Data TLB] ▶ [Loop FISSION 1.08 / Cache conflict] ▶ [Loop INTERCHANGE/Scalar Promotion 1.11 / Cache conflict] ▶ [Loop INTERCHANGE/Scalar Promotion 1.22 / Data TLB]	
	▶ [Loop FISSION 1.22 / Bandwidth] ▶ [Loop MERGING/INLINING 1.23 / FP FU resource] ▶ [Shifting 1.23 / Chain of dep. FP] ▶ [1.23]	1.23

Table 6: *Summary of optimization chains.*

gram transformation should be applied; this role is now devoted to dynamic analysis, as explained in previous sections. We believe this repartition will strike the right balance between static and dynamic analysis, and make it possible to automate the optimization process.

Automating the process will have several key benefits, especially for practical and industrial applications. (1) Many customer applications bear confidentiality issues so that people in charge of program tuning at the machine vendor have only access to a restricted part of the application, a restricted kind of data sets, and sometimes access simply cannot be granted. An automatic version of the manual process usually performed by tuning engineers is a method for bringing that expertise to codes with confidentiality constraints, usually the most critical applications for the customer. (2) As the optimization process evolves (different or refined analysis, different suggestions of optimizations or new optimizations, specialized trees per application domain,...) the process can be updated remotely without accessing the target applications themselves. Moreover, the customer can quickly benefit from the empirical expertise built at the machine vendor, program tuning becoming more like a recurrent service then a pre-sell activity. (3) It will become possible to apply the optimization process to all program sections and to avoid restricting generalization to a few code sections because of manpower constraints. (4) Finally, several customers can have large clusters of workstations where they distribute many jobs for one or a few applications. The process can be used to progressively finetune the application over the many executions, finding the program optimizations that realize the best tradeoff over the most frequent data sets, i.e., a form of continuous optimization [7].

6. CONCLUSIONS AND FUTURE WORK

We have presented a manual but systematic process for optimizing applications using extensive dynamic analysis, and we have demonstrated its practical application on a set of benchmarks. We are currently investigating several extensions to this work. First, we are trying to find whether we can extract enough information from hardware counters to guide (and speed up) the process, and at the same time we are experimenting with fast simulation techniques. Second, we are working on implementing a program transformation framework that facilitates the composition of long sequences of program transformation. Third, we are working on automating the process, especially by replacing manual code inspection with static analysis to find all possible opportunities of application of a large array of program transformations; we will then let the iterative process (and dynamic analysis) select the most appropriate transformation. Fourth, we want to carry on populating the decision tree and the mapping between program transformations and performance anomalies with new programs, i.e., optimization cases, and possibly start specializing the process by application domain. Finally, we want to apply the process to other processor platforms, especially x86 where internal micro-instructions make it difficult to predict the impact of a program transformation on the behavior of the architecture; a process that probes the appropriate transformations using dynamic analysis is particularly well suited for such platforms.

7. REFERENCES

[1] C. Bastoul, A. Cohen, S. Girbal, S. Sharma, and O. Temam. Putting polyhedral loop transformation to work. In *10th*

International Workshop on Languages and Compilers for Parallel Computing (LCPC), October 2003.

[2] K. D. Cooper, D. Subramanian, and L. Torczon. Adaptive optimizing compilers for the 21st century. *J. of Supercomputing*, 2002.

[3] J. Dean, J. Hicks, C. Waldspurger, W. Weihl, and G. Chrysos. ProfileMe: Hardware support for instruction level profiling on out-of-order processors. In *In Proceedings of the 30th International Symposium on Microarchitecture*, NC, Dec. 1997.

[4] G. Fursin, M. O'Boyle, and P. Knijnenburg. Evaluating iterative compilation. In *11th Workshop on Languages and Compilers for Parallel Computing*, LNCS, Washington DC, July 2002. Springer-Verlag.

[5] S. Girbal, G. Mouchard, A. Cohen, and O. Temam. DiST: A simple, reliable and scalable method to significantly reduce processor architecture simulation time. In *Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'03)*, San Diego, California, June 2003.

[6] Intel Itanium2 processor reference manual for software development and optimization. `http://developer.intel.com/design/itatium2/manuals`.

[7] T. Kistler and M. Franz. Continuous program optimization : a case study. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2003.

[8] T. Kisuki, P. Knijnenburg, M. O'Boyle, and H. Wijshoff. Iterative compilation in program optimization. In *Proc. CPC'10 (Compilers for Parallel Computers)*, pages 35–44, 2000.

[9] K. S. McKinley and O. Temam. A quantitative analysis of loop nest locality. In *ACM Symp. on Architectural Support for Programming Languages and Operating Systems (ASPLOS'96)*, 6, pages 94–104, 1996.

[10] Oprofile project. `http://oprofile.sourceforge.net`.

[11] Open research compiler. `http://ipf-orc.sourceforge.net`.

[12] D. Parello, O. Temam, and J.-M. Verdun. On increasing architecture awareness in program optimizations to bridge the gap between peak and sustained processor performance? matrix-multiply revisited. In *SuperComputing'02*, Baltimore, Maryland, Nov. 2002.

[13] E. Perelman, G. Hamerly, M. Biesbrouck, T. Sherwood, and B. Calder. Using simpoint for accurate and efficient simulation. In *Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'03)*, San Diego, California, June 2003.

[14] Perfmon project. `http://www.hpl.hp.com/research/linux/perfmon`.

[15] Standard performance evaluation corporation. `http://www.spec.org`.

[16] M. Stephenson, S. P. Amarasinghe, M. C. Martin, and U.-M. O'Reilly. Meta optimization: improving compiler heuristics with machine learning. In *ACM Symp. on Programming Language Design and Implementation (PLDI'03)*, pages 77–90, San Diego, California, 2003.

[17] Intel VTune performance analysers. `http://www.intel.com/software/products/vtune`.

[18] R. Wunderlich, T.Wenisch, B. Falsafi, and J. Hoe. Smarts : accelerating microarchitecture simulation via rigorous statistical sampling. In *In Proceedings of the 30th International Symposium on Computer Architecture*, San Diego, California, June 2003.

[19] K. Yotov, X. Li, G. Ren, M. Cibulskis, G. DeJong, M. Garzaran, D. Padua, K. Pingali, P. Stodghill, and P. Wu. A comparison of empirical and model-driven optimization. In *ACM Symp. on Programming Language Design and Implementation (PLDI'03)*, San Diego, California, June 2003.