Software refactoring guided by multiple soft-goals

Yijun Yu, John Mylopoulos, Lin Liu, Erik D'Hollander, Kristof Beyls

14th June 2003

Abstract

The software refactoring process is intended to enhance the quality of a software, by improving its understandability, performance etc. We adopt the framework of [24]in order to model and analyze software qualities, to determine what software refactoring transformations are most appropriate. In addition, we use software metrics in order to evaluate the qualities of a software system quantitatively. Our framework adopts and extends work reported in [27].

1 Introduction

Martin Fowler [10] defines software refactoring as "the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure". "Improvements to its internal structure" amount to improvements to the quality of the software system (also known as non-functional requirements). Examples of such improvements are "making the code easier to understand and cheaper to modify". Fowler's refactoring framework was proposed mainly for improving understandability and modifiability. However, the idea can also be applied to other qualities [24], such as performance, security, usability and more. In this work we focus on optimization transformations as a refactoring process for improving performance.

Software optimizers often focus on a single quality – performance. In software engineering practice, however, performance is just one software quality to be refactored. In most cases, other desired qualities – such as maintainability, reusability, robustness, testability, code simplicity, safety and privacy – maybe as important as performance, and may even have higher priorities.

In reality, not all soft-goals can be fully satisfied at the same time. For example, to improve performance, the code often becomes unnecessarily complex and therefore harder to maintain, or vice versa. Generally, trade-offs are often required when the qualities of interest are conflicting. The task of finding a "good enough" solution to the satisfaction of a set of qualities is actually similar to the problem of finding a Pareto-optimal solution in operation research [25, 17] or for the designers in CAD [12, 3]. In such a setting, the degree of satisfaction of a quality is best defined in terms of constraints, which can be further quantified using probabilities, fuzzy sets, or cost functions.

In this study, we adopt the framework proposed in [24] for modeling and analyzing software qualities. In this framework, qualities and the factors that affect them are modelled in terms of *soft-goals*. Moreover, the factors that affect each quality are represented in terms of a soft-goal interdependence graph (SIG). For this work, we focus on performance and code complexity by first surveying a number of optimizing transformations and several related complexity metrics, and then confirm their effectiveness through a clustering analysis using a number of metrics for benchmark programs. By associating metrics to soft-goals, it is possible to pinpoint conflicts among soft-goals. This identification is helpful in trade-off determination. Using a case study, we show how an incomplete SIG can be used to satisfy top-level soft-goals towards either or both performance and code simplicity.

2 The software refactoring model

According to Fowler [10], the refactoring process is composed of a number of small transformation steps: "while each refactoring step is simple, yet the cumulative effect of these small changes can radically improve the design". This is one of the keys to the success of the software refactoring process. However, we need a formal framework for selecting among many possible small steps, also to determine

its effect on different qualities.

Let P denote a set of programs or program units that are the subject of the refactoring and let R be the real number set. Then $X = \{x : P \rightarrow R\}$ denotes a set of quantitative metrics that measure things such as clockticks, number of instructions, number of memory access instructions, number of misses at L1 or L2 cache, McCabe complexity metrics, Halstead metrics of length and volume.

Let $T = \{t : P \rightarrow P\}$ be a set of refactoring transformations. also, let $G = \{g : P \rightarrow [0, 1]\}$ be a set of soft-goals for evaluating a program to some degree, thus they can be evaluated based on associated quality metrics $Q = \{q : \{x(p)|p \in P\} \rightarrow [0, 1]\}$.

Finding the best transformation for a single soft-goal associated quality metric q can be done by comparing the measured value of q. For each soft-goal $g \in G$ and its associated quality metric $q \in Q$, there are a set of positive transformations $T_+(q) \subset T$ and a set of positive programs $P_+(q) \subset P$ such that $q(x(t(p))) < q(x(p))$ for $t \in T_+(q), p \in P_+(q)$. Similarly there are negative and irrelevant transformations for certain programs.

Multiple soft-goals, however, may have non-overlapping positive cases, in other words, there is a possible transformation t such that $t \in T_+(q_1) \wedge t \in T_-(q_2) \vee t \in T_-(q_1) \wedge t \in T_+(q_2)$.

From n soft-goals, a quality space $[0, 1]^n$ can be formed by their associated quality metrics[1]. Each program is thus projected to a point in this space, while a transformation is a vector moving one point of the program to another point of the transformed program in the space.

The satisfaction of soft-goals can be represented as meeting the constraints on the qualities. While the satisfaction of a single soft-goal is judged by limiting a segment of one associated quality metric values, the satisfaction of multiple soft-goals can be judged by confining a region in the quality space. We call this confined region as a *satisfiable* region[2]. When a program is already projected inside the satisfiable region, no refactoring is necessary. Otherwise, a sequence of transformations is necessary to bring the qualities of the program into this region.

In principle, any transformation can change a program into another program, corresponding to a vector that goes from one point to another in the quality space. Better vectors go to the satisfied region more directly, while worse vectors go to the opposite direction; still another set of vectors can improve parts of the soft-goals, while harming others.

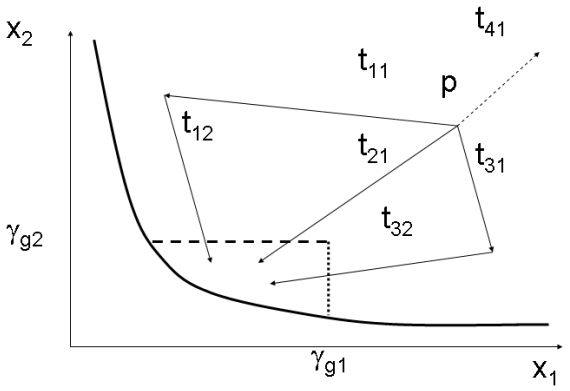


Figure 1: A view of refactoring process in the quality space. Two soft-goals are measured by qualities x_1 and x_2 and a satisfiable region is surrounded by $x_1 < \gamma_{g1}$ and $x_2 < \gamma_{g2}$ where $\gamma_{g1,g2} \in R$ are threshold related to the satisfice of the soft-goals. An implicit Pareto curve limits the satisfiability of both soft-goals. An original program is denoted as a point p outside the satisfiable region. Therefore transformations t_{ij} shown as vectors are considered with i being the number of alternatives at sequential steps j. There are four classes of transformations, for example, while t_{21} goes directly to the satisfiable region and t_{41} goes away; t_{11} and t_{32} satisfying g_1 but denying g_2; t_{31} and t_{12} satisfying g_2 while denying g_1. The three sequential applications of the transformations (t_{21}); (t_{11}, t_{12}) or (t_{31}, t_{32}) can lead to the satisfiable soft-goal region, thus are all possible refactoring solutions.

Because of the limitation in the available transformation as well as algorithms confined by the problem or functional requirements, the refactoring path often has to stop when reaching a physical Pareto-curve [26]. That means any direction or transformation can not lead to a better solution according to all objectives. Given solutions on the Pareto-curve, still one can weight them according to the priority of the soft-goals. Unlike Pareto-optimization, refactoring processes need not approach the Pareto-optimal curve if the soft-goal satisfiable region is already reached, they just need to be satisficed.

As quality metrics may depends on each other,

[1]Here $[0, 1]$ is adopt conveniently, one can use any ordinal region as long as the quality metric is comparable in consistent with the satisfactory of the soft-goal, that is, $x_1 < x_2$ if and only if $q(x_1) < q(x_2)$.

[2]So we can say that at this point that the soft-goals have been "satisficed" [24]

some soft-goals may depend on the satisfaction of others to be satisfied. A soft-goal g may depend on $g_1, \ldots, g_k$ if the quality metric associated with g is a function of the other quality metrics with $g_1, \ldots, g_k$. Even when the function is implicit, the soft-goal dependences can still be encoded in a hierarchical soft-goal interdependence graph(SIG) [24] qualitatively. In addition to the qualitative SIG, each soft-goal dependence carries a weight to determine how much does the parent soft-goals relies on the soft subgoal [11]. The weights can be very precise when the dependence link is known for the given program, they can also be imprecise when an average of possible values are given for a group of programs for a group of transformations.

In the Soft-goal Interdependence Graph (figure 2), nodes without out-degree are operational soft-goals, i.e., refactoring transformations. Among the transformations, there is also a hierarchy of cascading transformations.

Along with the SIG, a transformation decision tree can also extracted from the quality space view (figure 6). Starting from the original program p as the root of the decision tree, each applicable transformation leads to a new decision point for further selection of suitable transformations, until it proceeds to the satisfiable region. In refactoring guided by the SIG, we decompose the soft-goals into sub-soft-goals which are satisfiable by a subset of the transformations correspondingly. This process coincides with the decomposition of a transformation hierarchy that also limits the transformations to suitable ones. Since it is done incrementally, a satisfiability of all the soft-goals is the termination condition for the refactoring process.

3 Modelling performance and complexity

There are practical non-functional requirements to improve the speed of software (performance) and reduce the software complexity in code. Achieving the performance soft-goal reduces the operational cost while achieving the code simplicity soft-goal reduces the develop and maintenance cost [20].

For a given problem, an algorithm with less computational complexity can achieve higher performance with regard to the problem size; an implementation can do better than another if the computation resources (CPU, memory) are used more efficiently. This section establish a soft-goal interdependence graph for the possible conflicting higher performance and lower code complexity soft-goals and show the quantitative evidence for improving the time performance and reducing the code complexity.

3.1 Higher time performance

A decomposition of time performance soft-goal is shown in figure 2.

To achieve higher performance, both hardware and software improvements are useful. In the context of software refactoring, we consider mostly on the software improvements, while they improve because they make the programs fit better to the hardware platform. Therefore, the hardware possibilities such as faster CPU, multiprocessors, faster memory, larger cache with higher associativity are reflected by the refactoring techniques aiming to exploit the potentials. Therefore most soft-goals for software refactoring have corresponding hardware constraints as well. For example, more CPU required to implement parallelism, which can be improved by loop partitioning transformations [4, 7, 31]; enlarging cache size often resolve capacity cache misses while loop transformations like tiling and fusion helps to shorten the stack reuse distances [6] so as to avoid capacity misses for a given cache size [28, 23, 6], etc. Therefore in most cases, without considering the cost of hardware or the cost of software optimization, they are both positive alternatives to reach the performance soft-goal.

In order to obtain higher time performance, one needs to know where the time is spent in a program. There are basically three patterns of time distributions.

- **Sequential distribution.** Given two tasks run one after another sequentially, the total time spent is $T = T_1 + T_2$. For example, the program execution time can be decomposed into "computation" and "data communication" time. The former corresponding to the calculation involving ALU instructions, the latter involves memory access, I/O operations and bus/networka communication delay [14]. The distribution pattern can also applies to cumulative numbers relating to time. For example the total number of cache misses is the sum of capacity, conflict and cold misses, etc [1, 15].

- **Parallelism distribution.** Given two tasks that can run in parallel, the total time spent is $T =$

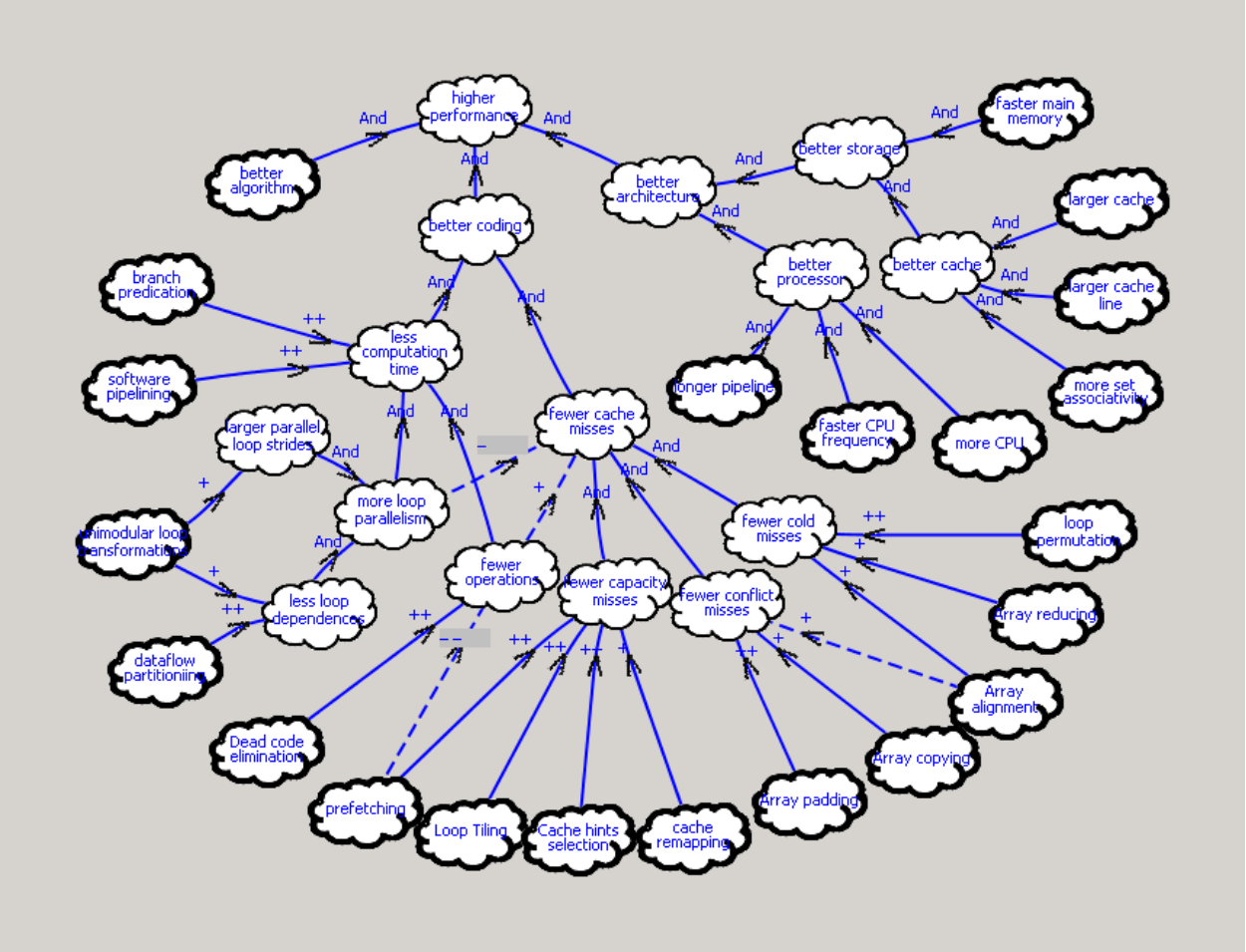


Figure 2: A soft-goal interdependence graph decomposed for time performance. The thinner cloud nodes are soft-goals while the heavier cloud nodes are operational soft-goals. The solid arrows reflect top-down decomposition of the soft-goal into "And", "Or" or supporting soft subgoals such as "Make(++)" and "Help (+)". The dashed arrows reflect correlational links between soft-goals on different decomposition paths to represent both positive and negative ("Hurt(−)/Break(−−)") contributions.

$max(T_1, T_2)$. For example, the parallel execution time depends on the critical dependency path in which the program can not finish until the longest subtask is done. The parallelism has different granularity such as tasks, loop iterations and instructions, they are classified as MIMD or SIMD respectively [9].

- **Redundancy distribution.** A task can be redundantly executed and the first finished execution will allow the completion of the task, thus the total time spent is $T = min(T_1, T_2)$. For example, the cache and main memory are two redundant units for memory access, the CPU looks for the data in the cache, if it is found, then the delay is cache access time, otherwise a much longer memory access time is payed for a cache miss. Reducing the ratio of cache miss will thus improve time performance [5, 14].

Each of the distributions has an associated metric such as time, number of misses, parallelism etc. which will be associated with a quality metric for a certain soft-goal, such as reducing time, reducing number of misses, increasing parallelism. Difference transformations helps each sub-soft-goal differently, meanwhile the time distribution gives weights to the sub-soft-goals to determine which category of transformations are better for achieving the soft-goal. This is known as the "80-20" percent rule. Identifying major players in the distribution (hot spots) helps to focus on suitable transformations that are more rewarding.

Many software optimization techniques were proposed by interactively studying benchmark programs [8], then were generalized into automatic compiler transformations [4]. Still quite some performance tuning transformations are not easy to achieve without good interactive programming tools [18, 32, 30]. In both automatic and interactive processes, the rationale of choosing suitable transformations are guided by soft-goals.

The use of compilers and programming tools alleviate the burden for code generation and transformation, however, the resulting code can be more complex for maintaining and understanding.

3.2 Less code complexity

Code complexity has effects on how difficult to test and to maintain, while good testability and maintainability lead to less defect rate. Thus it is an important soft-goal to reduce code complexity. In the software engineering literature, there are some code complexity metrics, such as McCabe's cyclomatic number $V(G) = e - n + 2$ [22], Halstead's software science metrics [13] or others. The Halstead's metrics n_1 and N_1 are the number of unique operators and total number of operators, n_2 and N_2 are the number of unique operands and total number of operands respectively; let $N = N_1 + N_2$ and $n = n_1 + n_2$, $L = N + n$ is the length of the symbol table, $N \log_2 n$ is the volume of the information content. Their interdependent relationships are shown in figure 3. According to experiments [20], these factors are the most related to the maintainability.

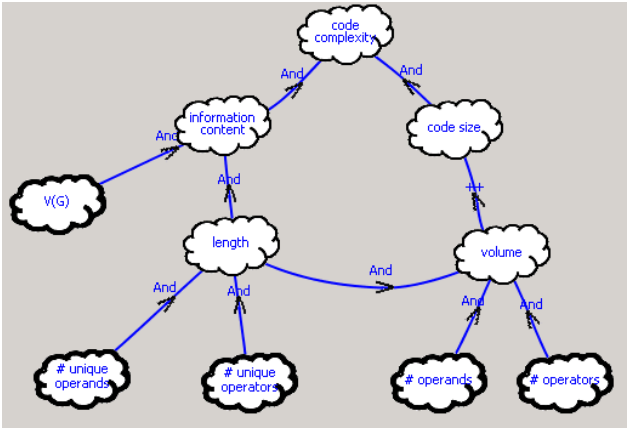


Figure 3: A SIG model for code complexity

Consider the "fewer operations" soft-goal in the performance SIG (figure 2). There are both positive and negative links correlational to "fewer operations". They are correlational because they are not direct evidence to support the parent soft-goals. However, once the link between "LOC" and "code size" and "information content" are certain in the code complexity SIG model, it is clear that some software optimization transformations do increase the code complexity metrics.

As far as various performance optimization techniques are concerned, hardware measures have almost no influence on the code complexity. However, some software techniques such as dataflow partitioning would severely increase the LOC. On the contrary, some techniques like dead code elimination could reduce the LOC. Loop transformations [4, 7, 2] and software pipelining [19] perform the same number of iterations with more control structures which requires only some book keeping statements (higher information contents). Still some software techniques like loop permutation [29] and

array padding [2] do not increase code complexity at all. Figure 4 presents the joint picture of the correlational links between software optimization techniques and the software code complexity metrics.

As quality metrics, we have chosen execution time ratio for the high performance soft-goal and the average of McCabe's and Halstead's complexity metrics ratio for the low complexity soft-goal. We have applied these quality metrics in an experiment performed on a small program. The advantages of our approach are two-fold, 1) the refactoring practice leads to gain experience in constructing and improving the SIG; 2) an established and reasonable SIG can be used to estimate metrics and guide refactoring practice in the long run.

4 Experiments

The experiments are given to show two possibilities: 1) how to setup a reasonable SIG model from the metrics; 2) evaluate the fulfillment of the soft-goals on the quality space quantitatively.

First we evaluated 10 metrics for the 10 programs in SPECfp95 benchmarks [8]. Each benchmark program forms a sample for the statistical clustering analysis. The first 5 metric vectors are gathered using the hardware counters on a 1.2GHz Pentium4 processor with 128MB memory; the last 5 metric vectors are gathered using Datrix[16] from the programs converted by F2C. The measure of each metric forms a vector of values from the 10 programs. We first normalized the elements of each vector with values range [a,b] into a vector with value range [0,1] by a linear function $f(x) = (x - a)/(b - a)$. The Euclid distance between two metric vectors are used in the clustering analysis. The resulting clusters distinguish the performance metrics and the complexity metrics and also shows which metrics are statistically closer in Euclid distance to each other, as shown in figure 5.

Once the SIG has been set up from experience, it can be applied to software refactoring process to select right transformations accordingly. As a starting point, we consider the following Fortran program for multiplying two matrices $A \in R^{m \times l}$ and $B \in R^{l \times n}$ into a matrix $C \in R^{m \times n}$.

```
real*8 A(512,512),B(512,512),C(512,512)
M = L = N = 512
do i = 1 , M
  do j = 1, L
    do k = 1, N
      C(i,k) = C(i,k) + A(i,j) * B(j,k)
```

To compare the code at the same abstraction level, we do not include the simplest code using a single matrix multiplication operator in Matlab language; also we do not consider the fastest code such as calling an assembly library function.

In our experiment, we also do not consider hardware changes which have no direct impact on software complexity. Here we list the hardware parameters of the experiment computer: 1.2GHz Intel Pentium 4-M CPU which has a 8KB 4-way associative L1 cache with 64 bytes per cache line and a 512 KB 8-way associative L2 cache with 32 bytes per cache line; the main memory has 128MB. The compiler being used for generating executable is GNU G77 3.2 with -O3 optimization options.

Using VTune, it shows that "cache misses" is a bottleneck for performance because the L2 miss over number of load instructions is very high. However, the measurement are not enough to decide which refactoring should be made. Another factor to consider in refactoring is the multiple soft-goals, the general aim of the refactoring. For example, the aim can be expressed as "apply transformations to speedup the program 20 times without sacrificing the code complexity 4 times". One needs to consult a refactoring SIG like the one in figure 4.

In the first step of refactoring process, we consider what operational soft-goals are applicable. Fixed hardware platform and algorithm already limit us to the "better code" subcategory in figure 2. Initially we consider five applicable transformations to this program.

Loop unrolling Hardware configuration constraints applicable transformation. For example, although there are abundant potential parallelism for loop I and J as visualized in [32], we do not need to consider loop parallelization. On the other hand, loop unrolling helps scheduling the iterations in a pipeline inside the CPU. For example, unrolling the I-loop leads to the following program.

```
real*8 A(512,512),B(512,512),C(512,512)
do i = 1, M, 4
 do j = 1, L
  do k = 1, N
   C(i,k) = C(i,k) + A(i,j) * B(j,k)
   C(i+1,k) = C(i+1,k) + A(i+1,j) * B(j,k)
   C(i+2,k) = C(i+2,k) + A(i+2,j) * B(j,k)
   C(i+3,k) = C(i+3,k) + A(i+3,j) * B(j,k)
```

It performs better than the original code also because of better spatial locality reduces cache misses. However, as a result of loop unrolling, the software

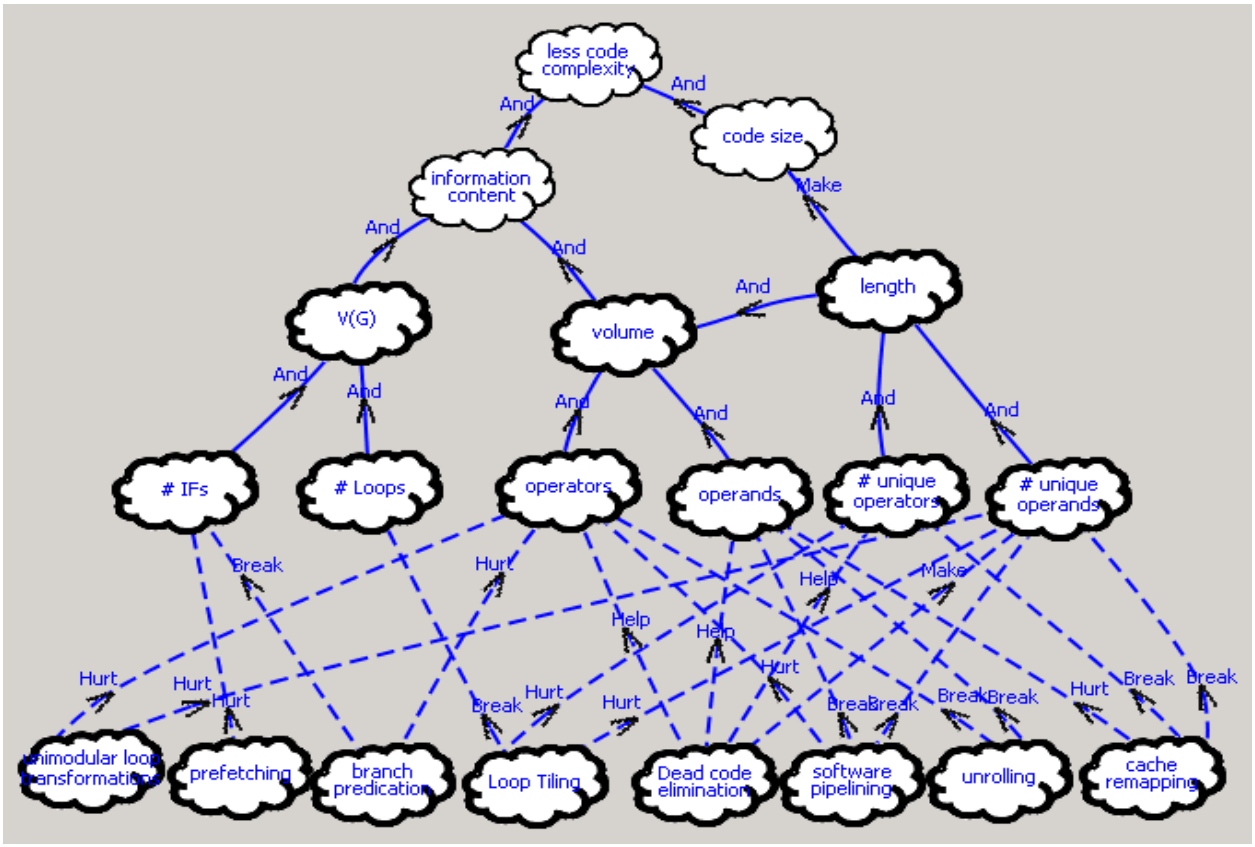
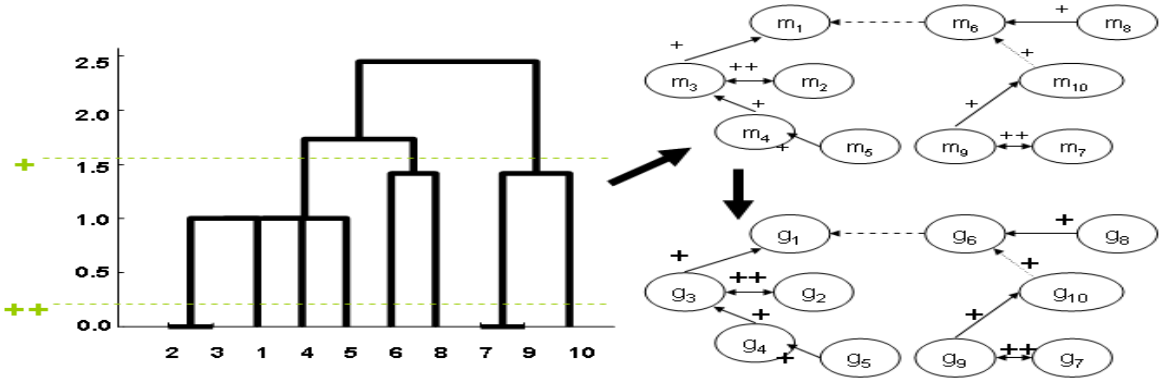


Figure 4: This combined SIG correlates some operational soft-goals in figure 2 with those in figure **??**.



Clustering the 10 quality metrics for SPEC95 benchmarks to verify the SIG model

m_1 = clockticks
m_2 = instructions
m_3 = load instructions
m_4 = L1 load instructions
m_5 = L2 load instructions
m_6 = v(G)
m_7 = operators (N_1)
m_8 = operands (N_2)
m_9 = unique operators (n_1)
m_{10} = unique operands (n_2)

Figure 5: The cluster analysis of metrics measurements derives a SIG.

complexity increases because additional assembly instructions were introduced which violates the low complexity soft-goal.

Loop tiling Loop tiling helps to resolve cache misses bottleneck, for example, three additional loops can be constructed to make the innermost loops concentrating on calculation.

```
real*8 A(512,512),B(512,512),C(512,512)
do i = 1, M, B1
 do j = 1, L, B2
  do k = 1, N, B3
   do ib = i, min(i+B1, M)
    do jb = j, min(j+B2, L)
     do kb = k, min(k+B3, N)
  C(ib,kb) = C(ib,kb)+A(ib,jb)*B(jb,kb)
```

Here `B1, B2, B3` are called blocking or tile size. The tile size selection is followed by the algorithm in [28, 23].

After the tiling, however, three additional loops added to the complexity of code.

Array padding and loop permutation Array padding is to enlarge the size of the array declaration without touching the control code, loop permutation is to change the nesting order of the loops. They can be both applied here.

```
real*8 A(515,515),B(515,515),C(515,515)
do k = 1, N
 do j = 1, L
  do i = 1 , M
   C(i,k) = C(i,k) + A(i,j) * B(j,k)
```

They are good for performance because padding an array can separate stack reuse distance farther from the power of two; and permutate the loop nest order from (i,j,k) to (k,j,i) can shorten the stack reuse distance and better exploit spatial locality. Meanwhile, the code has the same simplicity as the original code.

Register allocation The register allocation put a temporary variable in register to simplify the calculation.

```
real*8 A(512,512),B(512,512),C(512,512)
real*8 TA, TC
do i = 1 , M
 do j = 1, L
  TA = A(i, j)
  TC = 0
  do k = 1, N
   TC = TC + TA * B(j,k)
  C(i,k) = C(i,k) + TC
```

Besides the above transformations, in experiment we also investigated the impact of different optimization techniques such as dynamic memory allocation, dot-product.

Table 4 shows the measured results for the programs after applying individual and combined transformations.

Using the above experiment data, the log scale view of the quality space of time and complexity indices are shown in figure 7.

There are techniques that improves one but severely do harm to the other, while other techniques provides net improvement to both soft-goals. The decision maker can choose the one suited for the intention. Figure 6 shows five major transformations as decision making alternatives. It is based on the initial soft-goals as "apply transformations to speedup the program 20 times without sacrificing the code complexity 4 times", which is indicated as shadowed region in figure 7.

After go-through of the performance soft-goal, one can decide four transformations except for register allocation are among the first choices. The register allocation does not show good because for simple code, a compiler has done the job. As shown later when the code becomes more complex, even the compiler can not do register allocation well, so register allocation will be reconsidered in alter steps. After considering the complexity sub-goal, one can rule out unrolling because it does harm to complexity.

As shown in figure 7, none of them meet all the above soft-goals at once. One can further explore the possibilities of further transformations until all of them are satisfiable. In the second step, one can consider the array padding, loop permutation and loop tiling further to the previous transformations. Note after loop tiling, loop unrolling and register allocation becomes possible to choose because they do less harm to the complexity of already tiled program.

5 Related work

The software refactoring explained by Martin Fowler [10] emphasizes understandability and maintainability which is one of the non-functional requirements. In his book, performance is already an issue which has been delegated to a separate optimization process. Because an optimization process can also be traced by a sequence of small transformation steps, we combine it with the maintainabil-

Table 1: For the programs after the numbered transformations and the composition of these transformations, performance metrics such as clock-ticks, instructions, loads, L1, L2 cache misses are measured using Intel VTune, and the code complexity metrics such as McCabe's V(G), Halstead's length and volume are gathered from the C programs converted by *f2c* using *AT&T Datrix*. The time ratio is derived from the clock-ticks, the complexity ratio is derived from the average of three ratios of V(G), length and volume in comparison to the original program: $C(i) = (\frac{VG(i)}{max\{VG(i)\}} + \frac{volume(i)}{max\{volume(i)\}} + \frac{length(i)}{max\{length(i)\}})/3$. The last two columns are plot at log scale in figure 7.

program	clockticks	v(g)	length	volume	time ratio	complexity
t0: original	72459721152	4	106	530	1.0000	1.0000
t1 :padding(515,515)	6831847185	4	106	530	0.0943	1.0000
t1':padding(525,512)	6632600832	4	106	530	0.0915	1.0000
t2:permutation(k,j,i)	1430998290	4	106	530	0.0197	1.0000
t3 :tiling (18,17,20)	5013512820	7	195	1,045	0.0692	1.8536
t3':tiling(104,104)	3125096604	6	166	877	0.0431	1.5738
t3":tiling (12,12,12)	5781810670	7	195	1,045	0.0798	1.8536
t3"':tiling (5,5,5)	9446102100	7	195	1,045	0.1304	1.8536
t4i:unroll4(i)	17564822896	4	233	1,195	0.2424	1.8177
t4j:unroll4(j)	17570702513	4	233	1,195	0.2425	1.8177
t4k: unroll4(k)	73102904927	4	233	1,195	1.0089	1.8177
t5:register allocation	72590936880	4	111	560	1.0018	1.0345
t6:dynamic alloc.	5011374152	4	161	766	0.0692	1.3211
t7:dot prod.	27194780968	5	197	1,088	0.3753	1.7205
t1,t2	1321067280	4	106	530	0.0182	1.0000
t1,t2,t3,t4k	981540803	7	298	1,637	0.0135	2.5497
t1,t2,t3,t4k,t5	1112412480	7	276	1,525	0.0154	2.4101
t3,t4k	4723355976	7	322	1,779	0.0652	2.7145
t4i, t4j	4746177810	4	774	3,970	0.0655	5.2642
t4k, t4k	72670252644	4	737	4,071	1.0029	5.2112
t1',t2,t3',t4i,t4j,t5	677087540	8	1117	6,752	0.0093	8.4255
t1',t2,t3,t3',t4i,t4j,t5	985466160	15	1928	12,485	0.0136	15.1652

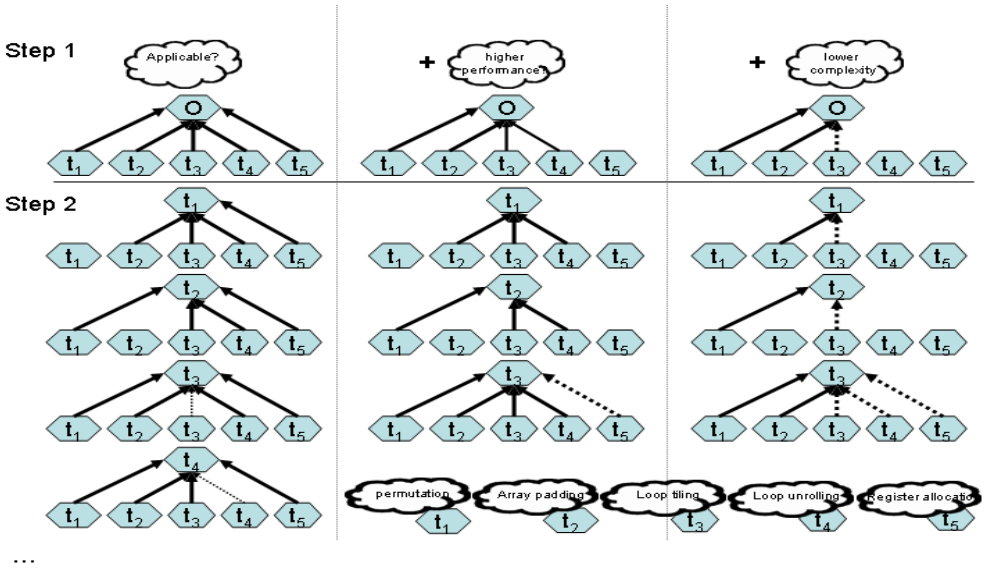


Figure 6: Two steps of refactoring are shown. Each step first identify the applicable tasks (operational soft-goals) for the given program (context), then decide which tasks are appropriate for performance and complexity. The weight given to a branch results from multiplying the distribution weight through measurement with the soft-goal dependence weight through quantified SIG.

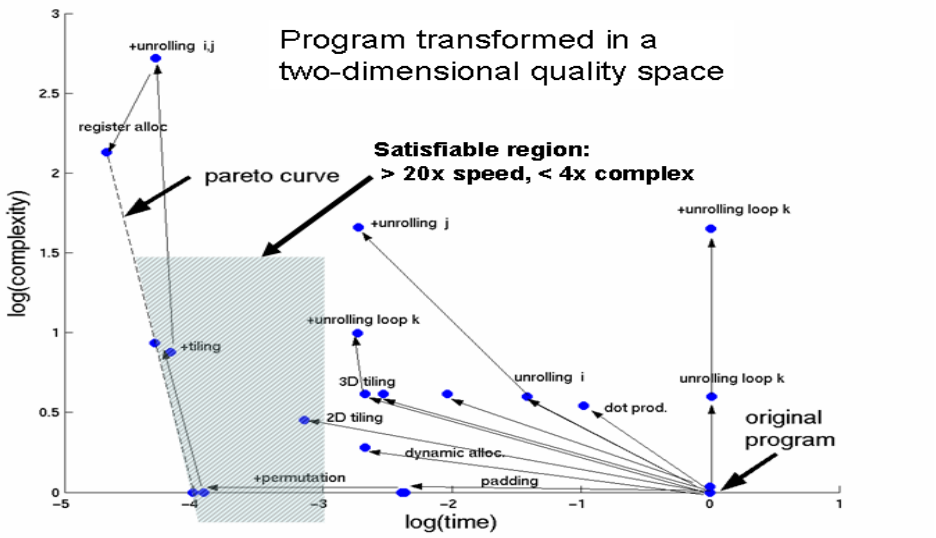


Figure 7: For the example programs, the quality space with time performance and code complexity indices are shown at log scale. Each program in table 1 is projected to a point in the space, each transformation produces an arrow from the program to another. It is clear that array padding, loop permutation, tiling and unrolling are effective optimizations when used properly, however, the last two increase complexity.

ity refactoring process and propose multi-soft-goal refactoring for non-functional requirements.

The non-functional requirement framework proposed by John Mylopoulos [24] is a reasoning facility to find supporting evidence for a given non-functional requirement soft-goal under functional requirement constraints. It was provided as a qualitative way of representing the incomplete knowledge of the non-functional requirements through soft-goals. The quantitative extension of the framework has been proposed either using Bayesian probability model [11] or Zadeh's fuzzy set model [21].

Ladan Tahvildari et al's work [27] first applied the NFR reasoning for comparing performance and maintainability of OO software in judging design patterns. It emphasizes more on the comparison of design patterns on maintainability issue, while this paper focus more on the performance tuning issue. In order to make the trade-off between the two issues clear, we plot the metrics in an quality space so that the refactoring process can be traced as one of the paths going to the satisfiable region.

One of the most studied problem in multi-objective or multi-criteria decision making is how to find non-inferior or Pareto-optimal solutions as multi-objective optimization requires [26]. When the underlying cost functions are implicit and the quality space is huge, it is not an easy task to find all the non-inferior solutions. In our case, finding a satisfiable solution in the refactoring process takes less computation than finding the non-inferior set (or Pareto-curve in 2D cases) because the search may terminate quicker. Since refactoring is an iterative incremental methodology, a non-optimal solution can be satisfiable already at a certain refactoring phase. The small steps philosophy makes it more flexible to the requirement changes.

For the successful maintainability refactoring process, the tool support is desirable to carry out tedious simple tasks for each refactoring step. For the successful performance refactoring process, a compiler is necessary to carry out tedious program transformations. Therefore an automated tool is ideal for the refactoring. However, it is still believed for more than a decade that compiler transformations need to be guided by interactive intuition of programmers [18] with the support of programming tools [32, 30] because the heuristics and analysis ability are conservative and ignoring possibilities that lies in the better understanding of the program. In addition, when performance is not the only or the primary concern, other qualities of software such as code complexity need to be addressed as well.

6 Conclusion

In this work, we have encoded the heuristics of performance and code complexity optimization as reasoning rules. This encoding was implemented in the NFR soft-goal interdependency graph model. The encoded knowledge are combined with the metrics measured by external profiling and measuring tools in order to guide the software refactoring processes with multiple soft-goals in mind. The case study of the matrix multiplication has shown that the decisions of software refactoring guided by graphically encoded soft-goals are less ad-hoc than those usually performed by software refactoring engineers. In the future, we will apply the soft-goal guided refactoring tool to large-scale softwares.

7 Acknowledge

We would thank Julio Cesar Leite for providing his useful insights to this paper.

References

[1] A. Agarwal, M. Horowitz, and J. Hennessy. An analytical cache model. *ACM Transactions on Computer Systems*, 7(2):184–215, May 1989.

[2] D.F. Bacon, S.L. Graham, and O.J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, DEC 1994.

[3] Richard Balling. Pareto sets in decision-based design. *Journal of Engineering Valuation and Cost Analysis*, 3:189–198, 2000.

[4] Utpal Banerjee, Rudolf Eigenmann, Alexandru Nicolau, and David A. Padua. Automatic program parallelization. *Proceedings of the IEEE*, 81(2):211–243, feb 1993.

[5] L. A. Belady. A study of replacement algorithms for a virtual storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[6] K. Beyls and E. H. D'Hollander. Reuse distance-based cache hint selection. *Lecture Notes in Computer Science*, 2400:265–??, 2002.

[7] Erik H. D'Hollander. Partitioning and labeling of loops by unimodular transformations. *IEEE Transactions on Parallel and Distributed Systems*, 3(4):465–476, jul 1992.

[8] K. M. Dixit. The SPEC benchmarks. *Parallel Computing*, 17(10–11):1195–1209, December 1991.

[9] M.J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, 21(9):948, 1972.

[10] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[11] Paolo Giorgini, John Mylopoulos, Eleonora Nicchiarelli, and Roberto Sebastiani. Reasoning with goal models. *Lecture Notes in Computer Science*, 2503:167–??, 2002.

[12] Tony Givargis, Frank Vahid, and Jorg Henkel. System-level exploration for pareto-optimal configurations in parameterized systems-on-a-chip. In *ICCAD*, pages 25–30, 2001.

[13] Maurice H. Halstead. *Elements of Software Science*. Elsevier North-Holland, New York, 1 edition, 1977.

[14] John L. Hennessy and David A. Patterson. *Computer Architecture – A Quantitative Approach*. Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, third edition, 2002.

[15] M.D. Hill and A.J. Smith. Evaluating associativity in CPU caches. *IEEE Transactions on Computers*, 38(12):1612–1630, Dec 1989.

[16] Richard C. Holt, Ahmed E. Hassan, Bruno Lague, Sebastien Lapierre, and Charles Leduc. E/r schema for the datrix c/c++/java exchange format. In *Working Conference on Reverse Engineering*, pages 284–286, 2000.

[17] Kasprzak. Pareto analysis in multobjective optimizationi using the colinearity theorem and scaling method. *Journal of Structural and Multidisciplinary Optimization*, 22(3):208–218, 2001.

[18] K. Kennedy, K.S. McKinley, and C.W. Tseng. Interactive parallel programming using the ParaScope editor. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):329–341, Jul 1991.

[19] M. Lam. Software pipelining: an effective scheduling technique for vliw machines. In *Proceedings of the SIGPLAN'88 conference on Programming Language design and Implementation*, pages 318–328. ACM Press, 1988.

[20] David L. Lanning and Taghi M. Khoshgoftaar. Modeling the relationship between source code complexity and maintenance difficulty. *Computer*, 27(9):35–40, September 1994.

[21] Jonathan Lee, Nien-Lin Xue, and Jong-Yih Kuo. Structuring requirement specifications with goals. *Information Systems*, 43(2):121–135, 2001.

[22] Thomas J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, December 1976.

[23] K.S. McKinley, S. Carr, and C.W. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, Jul 1996.

[24] John Mylopoulos, Lawrence Chung, and Brian Nixon. Representing and using nonfunctional requirements: A process-oriented approach. *IEEE Transactions on Software Engineering*, 18(6):483–497, Jun 1992. Special Issue on Knowledge Representation and Reasoning in Software Engineering.

[25] M. Sakawa, H. Yano, and J. Takahashi. Pareto optimality for multiobjective linear fractional programming problems with fuzzy parameters. *Information Sciences*, 63:33–53, 1992.

[26] Masatoshi Sakawa. *Large Scale Interactive Fuzzy Multiobjective Programming*. Physica-Verlag, Heidelberg, Germany, 2000. ISBN 3-7908-1293-5.

[27] Ladan Tahvildari and Kostas Kontogiannis. A workbench for quality based software re-engineering (doctoral symposium). In *Addendum to the 2000 proceedings of the conference on Object-oriented programming, systems, languages, and applications (Addendum)*, pages 157–158. ACM Press, 2000.

[28] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. *SIGPLAN Notices*, 26(6):30–44, 1991.

[29] Michael Wolfe. Parallelizing compilers. *ACM Computing Surveys*, 28(1):261–262, mar 1996.

[30] Y. Yu, K. Beyls, and E. D'Hollander. Visualizing the impact of the cache on program execution. In *5th International Conference on Information Visualization (IV '01)*, pages 336–341, Washington - Brussels - Tokyo, July 2001. IEEE.

[31] Y. Yu and E. D'Hollander. Partitioning loops with variable dependence distances. In *Proceedings of 2000 International Conference on Parallel Processing (29th ICPP'00)*, Toronto, Canada, August 2000. Ohio State Univ.

[32] Y. Yu and E.H. D'Hollander. Loop parallelization using the 3D Iteration Space Visualizer. *Journal of Visual Languages and Computing*, 12(2):163–181, April 2001.