8 How strictly should performance guarantees be expressed?

One fundamental problem we address here is, how can a software library standard be written so that

- library implementors have the freedom to take some advantage of hardware platform characteristics, yet
- application programmers have sufficient guarantees about the performance of library components that they can easily port their programs from one platform to another.

The solution we propose is **algorithm concept hierarchies**.

We are also envision such concept hierarchies as the best means to give compilation systems access to sufficiently accurate characterizations of algorithms.

9 Performance guarantees in the C++ standard

- The requirements are mostly stated in traditional O-notation, which suppresses constant factors and thus is incapable of distinguishing between two algorithms with the same asymptotic behavior but different constant factors.
- In some cases exact or approximate bounds are given for operation counts, of some principal operation (like comparison operations, in sorting algorithm descriptions).

10 A typical algorithm specification in the C++ standard

heapsort (Simplified from **partial_sort**'s description.)

Prototype:

```
template⟨class RandomAccessIterator⟩
void heapsort(RandomAccessIterator first,
              RandomAccessIterator last)
```

Effects: Sorts the elements in the range [first, last), in place.

Complexity: It takes approximately $N \log N$ comparisons, where $N =$ last $-$ first.

11 The challenge of generic software components

- Libraries like STL (and MTL, BGL, ...) provide generic components, which are designed with parameters representing infinite sets of abstractions (such as sets of types, which are represented in C++ for example by template parameters).
- For such generic components, machine instruction counts or memory accesses cannot be measured or derived analytically unless the component's parameters are all instantiated with specific types to produce a nongeneric instance.
- But each generic component has infinitely many such instances.

  The best we could do is try to measure or analyze the machine level statistics and catalog them for the "most common cases."

- Yet, while genericity seems to complicate matters, in fact it also contributes new ideas about how to express algorithm performance.

12 How generic programming can contribute to solving our problem

- In brief outline, the approach we are taking is:
  - Develop *algorithm concept hierarchies* similar to previously developed hierarchies for container and iterator concepts.
  - Use these algorithm concepts to present and organize performance requirements for a standard library's algorithm components.
  - Start with a way of expressing these requirements that is well matched to the level of abstraction of generic algorithms.
  - Extend it so that it takes into account key hardware characteristics such as cache size and speed.

13 Extensions

13.1 Why principal operation counting is not enough

- Looking only at principal operation counts ignores important hardware differences, in

  available instruction sets, number and speed of arithmetic units, registers, size and speed of caches and memory, etc.

- This abstractness can lead to suboptimal choices of algorithms for a particular task.

13.2 Extending principal operation counting

- How we might extend principal operation counting to take better account of hardware differences:
  - Express algorithms with additional parameters that **capture key hardware characteristics as a concept**, e.g., a cache concept.
  - Then study the performance of different algorithms or algorithm variants as assumptions about these parameters are varied—i.e., **are refined into different subconcepts**.

13.3 Organizing details and summarizing statistics

- Main drawback to introducing and varying hardware parameters: the **amount of detail** that must be reported to give a fully accurate picture of an algorithm's performance.
- But organization of information using concept lattices could help in
  - suppression of details at one level while revealing them fully at deeper levels;

    - summarization, aggregation of statistics.
    - providing a database of detailed performance statistics, corresponding to different hardware platforms, that can be accessed at both compile time and run time:
        * to help make the most appropriate choice of algorithms from a library depending on the specific context in which an computation is to be performed, and thus
        * to assist overall in optimizing applications for a particular platform.

14 Conclusion

14.1 Recap

- Concept lattices are a means to classify, present, and use knowledge of abstractions based on incrementally defined sets of requirements.
- Several applications of concept lattices we have found useful:
    - generic component library design,
    - high-level compiler optimizations,
    - library transformations,
    - algorithm performance specification, especially for generic algorithms

14.2 Next steps

- Refinement of algorithm concepts based on a variety of hardware parameters.
- Conceptual specification of distributed and parallel applications.

- Representation of concept-based information in a form that best supports the program analysis that the optimizing concept-based compiler performs.