This is an author produced version of a paper presented at [Conference name, date and location]. This paper has been peer-reviewed but may not include the final publisher proof-corrections or pagination.

Citation for the published paper:
Christoph Kessler and Welf Löwe
“Context-Aware Composition of Parallel Components”
1st International Software Technology Exchange Workshop,
Stockholm, November 23, 2011

Access to the published version may require subscription.
Context-Aware Composition of Parallel Components

Christoph Kessler
PELAB, Dept. of Computer and Inf. Science
Linköping University, Sweden
Christoph.Kessler@liu.se

Welf Löwe
Software Tech. Group at DFM
Linnaeus University, Växjö, Sweden
Welf.Lowe@lnu.se

ABSTRACT
We describe the principles of a novel framework for performance-aware composition of sequential and explicitly parallel software components with implementation variants. Context-aware composition dynamically selects, for each a performance-aware component, the expected best implementation variant, processor allocation and schedule for the actual problem size and processors available. The selection functions are pre-computed statically using machine learning based on profiling data.

1. INTRODUCTION
Software components are a well-proven means to organize complex software systems. They hide their implementation behind well defined functional interfaces that capture the possible functional interactions with their environment explicitly. Current component technology fits well the domain of sequential and concurrent object-oriented programming. However, existing component models and composition techniques are poorly suited for the performance-aware composition of components for parallel computing. Classical component systems allow composition of functionality beyond language and platform boundaries but disregard the performance aspect. The high performance computing domain and, due to recent trends in multi-core hardware, very soon even mainstream computing require composition systems with explicitly parallel components. In particular, a performance-aware component model and composition technique are necessary to build complex yet efficient and scalable software for parallel target platforms.

Conventional components hide and already bind design choices that ought to be selected only later in non-local optimizations. An example is the number for processors allocated for executing a parallel component. Such choices should better be decided after defining the components and even after composing them to a system, e.g. at deployment time or even at run time, when more information about the global execution environment or even the run-time context are available.

Earlier work in the auto-tuning domain has focused on such late internal adaptations at implicitly pre-defined program constructs such as loops that are optimized then by auto-tuning compilers and library generators. In this work, we make auto-tuning machinery available to component and component system programmers.

We present context-aware composition in Section 2. We exemplify this approach in Section 3 and, finally, we conclude and discuss directions of future work in Section 4.

2. CONTEXT-AWARE COMPOSITION
We propose a new approach for optimized composition of components that encapsulate sequential and explicitly parallel code. The component provider includes implementation variants which might be efficient in certain run-time contexts. These variants are hidden behind a compositional interface which unifies functional equivalent implementations. By annotating component implementation variants with performance-related meta-data such as a predictor for the expected execution time on the target platform, the provider makes the components performance-aware.

Equivalent functions are a natural variation point of possible auto-tuning adaptation. Individual implementations may mark up additional variation points at component calls, including the identification of independent subtasks for which a schedule and a resource allocation can be decided later instead of hard-coding such decisions for some particular target platform.

A predictor for the expected completion time on the target hardware could be given as a function of actual context attributes of a variation point, such as actual problem sizes and available resources for execution. Hence, the actual variants are only selected at deployment or even at execution time before each variation point is executed. At each such variation point, a dynamic dispatcher selects the expected fastest combination of the variants among equivalent components, the expected best schedule and resource allocation, etc.

The dispatcher is pre-computing statically, e.g., by computing dispatch tables, to minimize the run-time overhead of dynamic composition. It is based on the profiled runtime measured in test executions or deduced analytically, or combinations thereof. Based on the runtime of different variants in the different actual contexts, the dispatcher is constructed automatically using machine learning. More specifically, we learn a general classifier which, for each combination of relevant context attribute values—such as values of variables characterizing the problem size or the number of processors still available—selects the champion variant. To avoid profiling a combinatorial explosion of variants, the order in which learning is performed is of high importance. In principle, dispatchers for base components need to be learned first, before dependent components can be regarded.
Altogether our approach consists of three phases: (i) Component designers implement components with variation points. Sequential and explicitly parallel variants are defined manually, and further implementation variants can be generated, e.g., by automatically scheduling the execution for different numbers of processors. Component variants are hidden behind functional component interfaces. Additionally, context attributes, i.e., formal parameters that have an impact on performance, are exposed. (ii) Using performance profiles mapping actual contexts to performance data for the different variants, we learn general classifiers serving as dynamic dispatcher during run-time composition. (iii) By dynamically asking the dispatchers at variation points, we select the respective champion variant depending on the actual execution context.

3. EXAMPLE

The three steps are explained in an example component system shown in Figure 1. It consists of a number of components each containing an interface and alternative implementations as explained below.

3.1 Phase (i) — Component Design

The Search component finds the index of a value in a sorted array. The Merge Sorted component merges two sorted arrays into one sorted array. The Sort component sorts arrays. Finally, the Merge Unsorted component merges two unsorted arrays into one sorted array.

The components come in variants: Search can be implemented in a linear scan of the array or with logarithmic-time binary search. Merge Sorted can be implemented sequentially or in parallel. The latter variant splits the larger of the two arrays into two sub-arrays of half the size, uses Search to find the index $j$ in the smaller array $B$ such that $B[j] \leq A[i]$ and splits it there, and in parallel calls Merge Sorted on the smaller and larger parts of $A$ and $B$, respectively. There are many well known sequential and parallel variants of Sort; the parallel variants that we refer to recursively invoke Sort on the smaller problem sizes in parallel. Finally, Merge Unsorted may either concatenate the two arrays before a call to Sort or sort the input arrays before a call to Merge Sorted. The latter may sort the arrays sequentially one after the other or in parallel.

Note that we carefully avoided to expose an implementation variant beyond the component interface level: the implementation variant Merge Sort uses the Merge component, not one of its specific implementation variants; the Merge Unsorted implementation variants use Sort and (possibly) Merge Sorted and not one of their specific variants. This continues to hold for the recursive usages: Quick Sort and Merge Sort, resp., recursively call Sort and not an implementation variant thereof; Merge Parallel recursively calls Merge and not an implementation variant. In short, composition never explicitly selects an implementation variant.

Finally, component design needs to determine the context attributes which are relevant for performance. Besides the number of processors available, which is a default context attribute of parallel component implementations, we select the size of the array as context attribute.

![Figure 1: UML Diagram of our example component system. Interfaces are shown in bold italic, parallel and sequential implementations in black and gray, respectively.](image)

3.2 Phase (ii) — Learning the Dispatcher

First the call graph of the component system is determined in order to derive the order in which the different components are profiled and their dispatchers are learned. In our example, we profile and learn Search first, as it is a base component, then Merge Sorted, then Sort, then Merge Unsorted.

For each component, we generate test data varying the context attributes: in Search, arrays of random sorted values varying their size and a random value to search for. For the array size, we select sample values. Implementing training data generators varying the actual context attributes is actually part of component design; the generators may be redefined during composition or deployment, when more specific knowledge about actual problems (actual parameters) is available. It can even be postponed to runtime, where real problem instances are taken to compare the different variants.

For each actual context (sample) all variants are profiled and a champion is decided. This data is entered to machine learning, where a classifier interpolates and extrapolates a decision between the sample points.

Once a dispatcher for a component is ready, the depending components can be trained. In our example, Merge Sorted can be profiled and the dispatcher learned after the Search dispatcher is ready. It is essential to note that profiling of dependent components already uses the constructed dispatch-
In the sequential and parallel variants of Merge Sort (RAM).

In these cases, profiling and learning must interleave: We profile base cases (without recursion), learn the dispatcher for these cases, profile cases where the recursive call contexts have been profiled and learned already etc.

Similarly, we first profile and learn contexts with only one processor available, then we profile and learn contexts with more processors, which might be recursively reduced to the single processor cases which are then learned already.

### 3.3 Phase (iii) — Dynamic Composition

Before each variation point the connected dispatcher is invoked with the actual context attributes. It returns the expectedly best variant for this context. This dynamic composition can be seen as a generalization of the dynamic dispatch of object-oriented programming. In object-oriented programming, the actual dynamic type decides the method a call is bound to. The decision is implemented in a dispatch table. In context-aware composition, the actual context decides the variant (usually a method) a variation point (usually a method call) is bound to. The decision is implemented in a dispatcher, a trained general classifier (which could be a decision table as well).

The dispatcher allows to postpone the decision until runtime whether a parallel or a sequential variant should be used and which. In a certain actual context (e.g., only one core available) it might select a sequential variant; in another context (e.g., many cores still available and heavy work ahead) it is instead likely to select a parallel variant. For instance, on a 2 core machine Sort & Merge Parallel invokes Sort in parallel and the sort dispatcher will bind each of these parallel calls to a sequential Sort variant. In contrast, calls from Sort & Merge Sequential and Compose & Sort to Sort will (for sufficiently large arrays) be directed to a parallel Sort variant.

In the recursive compositions, e.g., in Merge Sort and Quick Sort and in Merge Parallel, the dynamic variant selection not only selects the appropriate processor allocation. It also find the break-even point in the problem size where an asymptotically worse algorithm should be selected because of smaller constants in its average case time complexity, for instance Insertion Sort or Linear Search. Figure 2 shows the performance of the performance-aware Sort component compared to the monolithic versions on a multi-core PC.

### 4. CONCLUSION AND FUTURE WORK

Context-aware composition enables system development by reusing parallel (and sequential) components. The design of these performance-aware components is (almost) as simple as the design of classic components. However, in contrast to classic components, performance-aware components postpone implementation decision such as resource allocation to runtime, which makes them way more efficient under composition.

We have validated this claim in a number of lab component systems based on sort & merge, matrix, and graph components and different machines up to 8 cores [1–6]. The next step is to validate it in an industrial context with a real-world component system.

### 5. REFERENCES


