# Optimized composition of performance-aware parallel components* **

Christoph W. Kessler[1] and Welf Löwe[2]

[1] IDA, Linköping University, 58183 Linköping, Sweden. chrke@ida.liu.se
[2] MSI, Linnaeus University, 35195 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. Automatic composition results in a table-driven implementation that, for each parallel call of a performance-aware component, looks up the expected best implementation variant, processor allocation and schedule given the current problem and processor group sizes. The dispatch tables are computed off-line at component deployment time by interleaved dynamic programming algorithm from time-prediction metacode provided by the component supplier.

## 1 Introduction

Software components are a well-proven means to organize complex software systems. Regardless if fine-grained or coarse grained, they hide their implementation behind a well defined functional interface that captures the possible functional interactions with their environment explicitly. However, conventional black-box components hide and hardcode too many design choices that are actually relevant for non-local optimizations and that should better be decided later, e.g. at deployment time or even at run time, when more information about the execution environment or the run-time context is available.

*Grey-box components* expose selected details of their internal functionality that are subject to late adaptation explicitly in a *compositional interface*. Grey-box composition includes classical caller-callee binding but also many further kinds of program adaptation and optimization, ranging from static application synthesis at deployment time to run-time adaptations. Functional interfaces alone can enable portability, but not performance portability.

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 propose leveraging grey-box composition to make auto-tuning machinery available to application level programmers.

We propose a new approach for optimized composition of grey-box components that encapsulate sequential or explicitly parallel code. This way they become *performance-aware*, i.e., the component provider (who knows the implementation details) includes performance-related meta-data and -code in an extended composition interface. This

* This paper is a revised, updated and extended version of our ParCo-2007 paper [9].
** To appear in: Proc. 15th Int. Workshop on Compilers for Parallel Computers (CPC-2010), Vienna, Austria, July 2010.

meta-data and code allows to predict, at deployment or even at execution time before each call, the expected completion time on the target system as a function of a run-time call context (such as problem sizes and available resources for execution). The compositional interface also declares functional equivalence of an implemented function with functions of other components, and marks up locations of possible auto-tuning adaptation in the component code. The latter includes the identification of independent subtasks for which the schedule and resource allocation can be decided later instead of hard-coding such decisions for some particular target platform.

We give an interleaved dynamic programming algorithm that computes off-line, from the provided performance meta-data and -code, dispatch tables for optimized component composition, and show how to transform the components at deployment time to inspect the tables at run time to direct the execution of the program. The table-driven execution will then, at any dynamic composition point such as a call to a performance-aware component function, select the expected fastest *combination* of the algorithmic variant among equivalent components, of the expected best schedule and resource allocation, etc. Due to static pre-computing of the dispatch tables, the run-time overhead in the generated table-driven code is low.

Using sorting as case study, we demonstrate for a simple simulated shared-memory parallel computer and for a sequential platform that our method can lead to considerable performance improvements compared to hard-coded composition choices.

## 2 Performance-aware components and interfaces

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 massively parallel computing. Classical component systems allow composition of functionality beyond language and platform boundaries but disregard the performance aspect. The high performance computing (HPC) domain and 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 highly parallel target platforms.

We propose a new approach for optimized composition of performance-aware sequential or explicitly parallel components. We require that all parallel and sequential components subject to performance-aware composition adhere to a specific *performance interface* recognized by a special *performance-aware composition tool*.

Performance-aware composition requires the component provider to supply meta-code for each performance-aware method $f$. We focus on terminating methods $f$; the meta-code is used to predict the execution time of $f$. The meta-code includes a `float`-valued function `time_`$f$ depending on the number $p$ of processors available and some selected parameters of $f$; `time_`$f$ approximates the expected runtime of $f$ used for composition and local scheduling purposes. It is supplied by the component provider (maybe but not necessarily based on a static analysis of $f$). In practice, it may interpolate and extrapolate entries in a precomputed finite table or use a closed form function or a combination of these techniques. As `time_`$f$ will be evaluated at runtime before

executing $f$, it should be computable quickly which prohibits, e.g., an implementation based on simulated execution.

Both functional and performance signatures are exposed by the component provider. Different substitutable component variants implementing the same functionality $f$ inherit from the same interface. In general, different variants have different `time_`$f$ functions with same signature. Dynamic composition chooses, for each call, the variant expected to be the fastest .

Parallel implementations may additionally exploit nested parallelism, marked explicitly by a parallel composition operator, putting together independent subtasks that could be executed in parallel or serially. Different schedules and processor allocations are precomputed, depending on static information like the characteristics of the hardware platform and the variants of subtasks available. They are then selected dynamically for different runtime configurations of problem sizes and processor group sizes. Hence, in the case of possible parallel implementations of $f$, the `time_`$f$ functions are also dependent on in the actual processor group size which is just another parameter.

## 3 Example: Parallel and sequential sorting components

In the following example, we use pseudocode with object-oriented syntax and a shared-memory programming model similar to Fork [8] where statements and calls are executed in SPMD style, i.e., by a group of processors synchronized at certain program points. Note that our framework is by no means restricted to SPMD or object-oriented paradigms. In particular, it should work equally well with modular languages that provide the interface concept. Moreover, it would work equally well with a message passing programming model such as MPI, as we exemplified for optimized resource allocation and scheduling at parallel composition of MPI components in earlier work [6].

We consider *sorting* as a first example of functionality supported by several parallel and/or sequential components. We picked sorting since alternative sequential and parallel sorting algorithms such as quicksort, mergesort, or bitonic sort are well-known. Moreover, there are variants highly depending on the actual input (quicksort) and others that are less input-independent (mergesort). Hence, sorting comprises properties of a broad range of potential applications. All component variants conform to the same interface:

```
/*@performance_aware*/
interface Sorting {
  /*@performance_aware*/
  void sort( float *arr, int n );
}
```

The `performance_aware` qualifier marks the interface and its method `sort` for the composition tool. It expects a performance-aware implementation variant of `sort` and a time function `time_sort` with all the parameters of `sort` in all implementation variants (but possibly only a subset of the parameters actually uses: we ignore `*arr` here). For instance, a parallel quicksort component implementing the `Sorting` interface is given in Figure 1.

```
/*@performance_aware*/
component ParQS  implements Sorting
                 requires Partitioning, Sorting
{
  float find_pivot( float *arr, int n ) { ... }

  /*@performance_aware*/
  void sort( float *arr, int n )
  {
     if (n==1) return;
     float pivot = find_pivot( arr, n );
     int n1 = partition( arr, n, pivot );
     /*@compose_parallel*/
         /*@1*/  sort( arr, n1 );
         /*@2*/  sort( arr+n1, n-n1 );
     /*@end_compose_parallel*/
  }

  /*@time_metadata    // parsed and used by composition tool
  // aux. for time_sort, found empirically at deployment time
  const float T_test_1 = ...      // time for recursion end
  const float T_find_pivot = ...  // time for find_pivot
  ...                 // micro-benchmarking meta-code omitted
  time_sort( int p, float * arr, int n )
  {
     if (n == 1) return T_test_1;
     float acctime=0.0;
     for (int n1=1; n1<n; n1++)
        acctime += get_T_p2_sort_sort( n1, n-n1, p );
     float exptime = acctime/(float)(n-1);
     return T_test_1 + T_find_pivot
                     + time_partition( p, arr, n) + exptime;
  }
  @end_time_metadata */
}
```

**Fig. 1.** A performance-aware parallel quicksort component (before composition).

The operator `compose_parallel` marks independent calls to the performance-aware method `sort`; these may be executed in parallel or serially, in any order. The composition tool will replace this construct with a dynamic dispatch selecting the approximated optimum schedule (serialization of calls, parallel execution by splitting the current group of processors into subgroups, or a combination of thereof) and implementation variant if several `sort` implementations are provided.

Within a `time_sort` function, the component designer specifies a closed formula, a table lookup, or a recurrence equation for the expected runtime of this variant. This meta-code is used to generate the dynamic dispatch tables. The operator `get_T_p2_sort_sort` yields the estimated time for the expected fastest parallel

```
/*@performance_aware*/
component SeqQS  implements Sorting
{
  /*@performance_aware*/
  void sort( float *arr, int n ) {
     seq qsort( arr, n );  // done on 1 processor only
  }
  /*@time_metadata    // used by composition tool
  const float[] T_qsort = ... // Table of seq. sorting times
  ...                 // micro-benchmarking metacode omitted
  time_sort( int p, float * arr, int n )
  {
     return T_qsort[n];
  }
  @end_time_metadata */
}
```

**Fig. 2.** A performance-aware sequential sorting component.

composition of two independent calls to `sort`, here with subproblem sizes `n1` and `n-n1` and the number `p` of processors available for executing the call. It is the approximation of the makespan of the schedule found in optimization, cf. Section 4.

Fig. 2 shows the `sort` and `time_sort` functions of a performance-aware sequential sorting component `SeqQS`, which simply wraps a (non-performance-aware) sequential quicksort library routine. For brevity, we omit the code of a parallel merge sort variant `ParMS` and parallel insertion sort `ParIS`, which are also used in our example implementation, cf. Section 5.

## 4 Performance-aware composition

The optimization problem for composition is to *simultaneously* determine ($i$) for each call to a performance-aware function, the (expected) best implementation variant, and ($ii$) for each parallel composition operator in a performance-aware function, the number of processors to spend on the calls of each subtask, and a schedule and resource allocation for these subtasks. We refer to this combined optimization problem as the *independent variant malleable task scheduling problem*.

A *malleable task* is a computational task that may be executed on $p = 1 \dots P$ processors. Its execution time is described by a non-increasing function $\tau$ in the number of processors $p$ actually used. A malleable task $t_f$ corresponds to a call to a performance-aware component function $f$, i.e., one specific implementation variant of functionality $f$. A (malleable) task calling functionality $f$ is called *variant* if there may be different implementation variants (components) $c$ for $f$, each being a (malleable) task $t_f^c$ with a different performance function $\tau_f^c$. The $\tau_f^c$-function is approximated using the `time_`$f$ function of component $c$. Then, `compose_parallel` is replaced by selecting a schedule of independent malleable tasks, each of which can be variant and thus have a different $\tau_f^c$.

Even without the choice between different variants, this scheduling problem is known to be NP-hard in the general case, but good approximations exist: for instance, $k$ independent malleable tasks can be scheduled in time $O(P \cdot k^2)$ on $P$ processors such that the completion time of the resulting schedule is at most $\sqrt{3}$ times the optimum [12]. Moreover, $k$ *identical* malleable tasks can be scheduled optimally to $P$ processors in time $O(\max(\log(k) \cdot t^{3P}, k \cdot (2t)^P)$ where $t$ is the sequential execution time of a task [5].

### 4.1 Dispatch table generation

The composition tool does not directly create the customized code. Instead, it generates $(i)$ a variant dispatch table $V_f$ for each interface function $f$ and $(ii)$ a schedule lookup table $S$ for each parallel composition operator, listing the (expected) best processor allocation and the corresponding schedule. The table lists entries with the best decision for a range of problem sizes (ranging from 1 to some maximum tabulated problem size, suitably compacted) and a number of processors (ranging from 1 to the maximum number of processors available in the machine, suitably compacted). For our example above, $V_{\texttt{sort}}[n][p]$ contains a pointer to the expected best `sort` function for problem size $n$ and processor group size $p$, cf. Fig. 6. $S_{\texttt{p2_sort_sort}}[n_1][n_2][p]$ for the parallel composition of two independent `sort` subtasks in `ParQS` yields a processor allocation $(p_1, p_2)$, where $p_1 + p_2 \leq p$, and a pointer to the expected best schedule variant; here it can only be one of two variants: parallel or serial execution of the two independent calls to `sort`.

The tables $V_f$ and $S$ are computed by an *interleaved dynamic programming algorithm* (see also Fig. 3) as follows. Together with $V_f$ and $S$, we will construct a table $T_f(n, p)$ containing the (expected) best execution times for $p$ processors.

For a base problem size, e.g. $n = 1$, we assume the problems to be trivial and the functions not to contain recursive malleable tasks, i.e., no recursive calls to performance-aware functions. Hence, $T_f[1][p]$ can be directly retrieved from the corresponding $\texttt{time}_f$ functions and $V_f[1][p]$ selected accordingly as the variant with minimum $\texttt{time}_f$.

For $p = 1$, parallel composition of independent subtasks always leads to a sequential schedule, i.e., the entries $S[n_1][n_2]\dots[n_k][1]$ encode a sequential schedule where each task uses 1 processor. Hence, no performance-aware function contains alternative schedules for $p = 1$ and `get_T_p2_sort_sort` reduces to a simple addition of the execution times of the subtasks. Accordingly, the $T_f[n][1]$ and $V_f[n][1]$ for $n = 1, 2, ...$ can be derived iteratively from the $\texttt{time}_f$ functions: $T_f[n][1]$ is set to the minimum $\texttt{time}_f$ of all variants of $f$ and $V_f[n][1]$ is set to the variant with minimum $\texttt{time}_f$. Usually, the sequential variants (not containing a parallel composition at all) outperform the serialized parallel variants.

Then, we calculate the remaining table entries stepwise for $p = 2, 3, ....$ For each $p$, we consider successively $n = 1, 2, ....$ For each such $n$, we determine $T_f[n][p]$, $V_f[n][p]$, and the schedules of the calls to malleable tasks $S_{...}[n_1][n_2]...[n_k][p]$. This is possible since, at this point in time, we have already computed $T_f[n'][p']$, $V_f[n'][p']$ of the subproblems with $n' < n, p' < p$ and their schedules.

First, we calculate the schedules $S_{...}[n_1][n_2]\dots[n_k][p]$ for the `compose_parallel` constructs: Since $T_f$ is defined for each call contained, we simply apply an

**Input:** Performance-aware interface function $f$ and several components for it;
maximum problem size $N$ and machine size $P$ to be tabled.
**Output:** Expectedly best variant dispatch table $V_f[1..N][1..P]$,
expected best time table $T_f[1..N][1..P]$,
expected best schedule/resource table $S_{\mathrm{p}d_f_1_..._f_d}[1..N]...[1..N][1..P][0..d]$
for each group of $d$ independent calls to performance-aware functions $f_1$,...,$f_d$
that occurs in some component for $f$.
**Method:**
<u>Base case 1:</u> base problem size (e.g., $n = 1$): no recursion.
for all $p = 1...P$,
obtain $T_f[1][p]$ and $V_f[1][p]$ directly from the minimum $\mathtt{time_}f(p, 1)$
for all of all registered components implementing $f$ and each $p$.
<u>Base case 2:</u> $p = 1$ :
Schedule for independent tasks: `compose_parallel` is always serial.
for all $n = 1, ..., N$
obtain $T_f[n][1]$ and $V_f[n][1]$ by minimizing over $\mathtt{time_}f(1, n)$
of all registered components implementing $f$.
<u>General case:</u>
For remaining columns $p = 2, 3, ...P$:
For problem sizes $n = 2, 3, ...N$:
NB: For all $n_1 < n$, $n_2 < n$, ..., $p_1 \le p$, $p_2 \le p$, ...:
Expectedly best variants $V...[n_1][p_1]$, $V...[n_2][p_2]$, ...
with times $T...[n_1][p_1]$, $T...[n_2][p_2]$, ... already computed.
Determine best schedule $S[n_1][n_2][p]$ for each `compose_parallel`
by approximation or brute-force-optimization over all $n_1 < n$, $n_2 = n - n_n 1$
Determine $T_f[n][p]$ and $V_f[n][p]$ by
1. Evaluating all $\mathtt{time_}f(p, n)$
with table-lookup of expected best schedule's makespan
(already computed) at each `compose_parallel`
2. Minimizing over all variants for $f$.

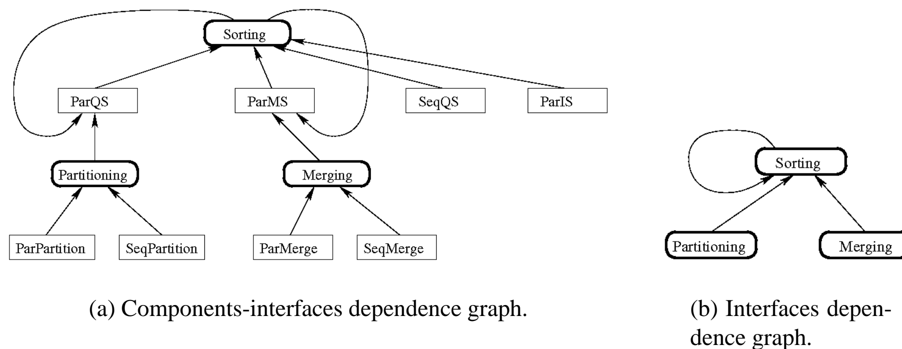
**Fig. 3.** The interleaved dynamic programming algorithm for computing the $V$ and $S$ tables.

approximation to the independent malleable task problem leading to a schedule and processor allocations $(p_1, p_2, \ldots, p_k)$, where all $p_i \le p$. In our example, we do not even need to approximate the optimum since there is only the parallel schedule with finitely many parallel allocations $(p_1, p - p_1)$ and the sequential one.

Second, we compute $T_f[n][p]$ and $V_f[n][p]$: We replace the `get_T_p2_sort_sort` construct with the makespan of the schedule derived and evaluate the $\mathtt{time_}f$ functions for each variant. Again, $T_f[n][p]$ is set to the minimum $\mathtt{time}_f$ of all variants of $f$ and $V_f[n][p]$ is set to the variant with minimum $\mathtt{time_}f$.

### 4.2 Order of processing the registered components

We represent the relations between components and interfaces in a bipartite directed graph, the *components-interfaces dependence graph* $G_{CI} = (C, I, D)$ where $C$ is the set of registered performance-aware components, $I$ the set of performance-aware interfaces, and $D \subseteq C \times I \cup I \times C$ the set of dependence edges such that $(c, i) \in D$




(a) Components-interfaces dependence graph.

(b) Interfaces dependence graph.

**Fig. 4.** The dependence graphs of our running example.

iff $c$ implements $i$, and $(i, c') \in D$ iff $c'$ requires interface $i$. See Fig. 4(a) for the components-interfaces dependence graph of our running example.

We can condense the components-interfaces dependence graph into a directed graph, the *interfaces dependence graph* $G_I = (I, E)$ with $E \subseteq I \times I$ such that $(i, i') \in E$ iff $\exists c \in C$ with $(i, c) \in D$ and $(c, i') \in D$. See Fig. 4(b) for the interfaces dependence graph of our running example. The interfaces dependence graph represents exactly the dependencies between interfaces for the dispatch table construction. All interfaces contained in a strongly connected component of $G_I$ must be processed together by our interleaved dynamic programming algorithm of Fig. 3, while disjoint strongly connected components of $G_I$ can be processed separately in topological order as their tables depend on each other in at most one direction.

### 4.3 Composition

Auxiliary performance functions and tables as needed for the $\texttt{time}_f$ functions are determined at component deployment time. Accordingly, a component variant provider needs to provide benchmark meta-code executed before optimization.

All performance-aware components for the same interface should be deployed together to get meaningful table entries. Encapsulation is still preserved as third-party component providers need not know about other performance-aware components that co-exist.

After optimization, the composition tool patches each call to $f$ in all component implementations with special dispatch code that looks up the variant to call by inspecting the $V_f$ table at runtime, and generates dispatch code at each parallel composition operator looking up its $S$ table at runtime with the current subproblem and group size to adopt the (expected) best schedule. The example code for `ParQS` after composition is sketched in Figure 5.

The table entry `S_p2_sort_sort[n1][n2][p][0]` contains the precomputed schedule variant, the entry `S_p2_sort_sort[n1][n2][p][i]` the processor allocation for the `i`-th call. `groupsize`() returns the number $p$ of processors in the executing group. Any other call to `sort(a,n)` would be patched to `V_sort[n][p](a,n)`.

```
component ParQS
{
  float find_pivot( float *arr, int n ) { ... }
  extern const int[][] V_sort;          // The V table
  const int[][][][] S_p2_sort_sort = ...;  // The S table
  void sort( float *arr, int n )
  {
     if (n==1) return;
     float pivot = find_pivot( arr, n );
     int n1 = partition( arr, n, pivot );
     // get remaining context data:
     int p = groupsize();  // number of executing processors
     // look up schedule and resource allocation in S table:
     int schedule = S_p2_sort_sort[n1][n-n1][p][0];
     int p1 = S_p2_sort_sort[n1][n-n1][p][1];
     int p2 = S_p2_sort_sort[n1][n-n1][p][2];
     switch( schedule ) {
        case 1:  // serialized schedule of 2 indep. sort calls:
                 V_sort [n1][p1] ( arr, n1, p1 ); // lookup V table
                 V_sort [n-n1][p2] ( arr+n1, n2, p2 );
                 break;
        case 2:  // parallel schedule of 2 indep. sort calls:
                 split_group(p1,p2) { V_sort[n1][p1](arr, n1); }
                                    { V_sort[n2][p2](arr+n1, n2); }
     }
  }
}
```

**Fig. 5.** The parallel quicksort component after composition (cf. Fig. 1). The brokering code injected by the composition tool is shown in blue color. For two independent calls to `sort()`, only two different schedules can occur.

Technically, the composition tool could be based on COMPOST [3] or a similar tool for static meta-programming that enables fully programmable program restructuring transformations. A prototype is currently being implemented.

## 5 Implementation and first results

A proof-of-concept implementation uses the C-based parallel programming language Fork [8] for three sorting components, parallel recursive quicksort, parallel recursive mergesort, and sequential quicksort. The time parameters for the functions `find_pivot`, `partition`, `qsort` etc. used in `time_sort` were obtained by measuring times for example instances. Times depending on input data were averaged over several instances. To record the best schedules for parallel composition in the $S$ table, a brute-force enumeration approach was chosen, as the best constellation of subgroup sizes can be deter-

```
V_sort P= 1 2 3 4 5 6 7 8  9      ...     16
       +---------------------------------
N=    1 :  3 3 3 3 3 3 3 3  3 3 3 3 3 3 3 3
N=    2 :  3 3 3 3 3 3 3 3  3 3 3 3 3 3 3 3
N=    3 :  3 3 3 3 3 3 3 3  3 3 3 3 3 3 3 3
N=    4 :  3 3 3 3 3 3 3 3  3 3 3 3 3 3 3 3
N=    5 :  3 4 4 4 4 4 4 4  4 4 4 4 4 4 4 4
N=    6 :  3 2 2 2 2 2 2 2  2 2 2 2 2 2 2 2
N=    7 :  3 2 2 2 4 4 4 4  4 4 4 4 4 2 2 2
N=    8 :  3 2 2 2 2 2 2 2  2 2 2 2 2 2 2 2
N=   10 :  3 2 2 2 2 2 2 2  2 2 2 2 2 2 2 2
N=   12 :  3 2 2 2 2 2 2 2  2 2 2 2 2 2 2 2
N=   14 :  3 2 2 2 2 2 2 2  2 2 2 2 2 2 2 2
N=   16 :  3 2 2 2 2 2 2 2  2 2 2 2 2 2 2 2
N=   20 :  3 2 2 2 2 2 2 2  2 2 2 2 2 2 1 1
N=   24 :  3 2 1 2 2 2 2 2  2 2 2 2 2 2 2 2
N=   28 :  3 2 1 2 2 2 2 2  2 2 2 2 1 2 2 2
N=   32 :  3 2 1 2 2 2 1 2  2 2 2 2 2 1 1 2
N=   40 :  3 2 1 2 1 1 1 1  1 2 1 2 1 1 1 1
N=   48 :  3 2 1 2 1 1 1 2  1 1 1 2 2 1 1 1
N=   56 :  3 2 1 2 2 1 1 2  1 2 2 2 1 1 1 1
N=   64 :  3 2 1 2 1 1 1 2  1 1 1 1 1 1 1 1
N=   80 :  3 2 1 1 1 1 1 1  1 1 1 1 1 1 1 1
...
N=  384 :  3 2 1 1 1 1 1 1  1 1 1 1 1 1 1 1
N=  448 :  3 1 1 1 1 1 1 1  1 1 1 1 1 1 1 1
N=  512 :  3 1 1 1 1 1 1 1  1 1 1 1 1 1 1 1
N=  640 :  3 1 1 1 1 1 1 1  1 1 1 1 1 1 1 1
N=  768 :  3 1 1 2 1 2 1 1  1 1 1 1 1 1 1 1
N=  896 :  3 1 1 1 1 1 1 1  1 1 1 1 1 1 1 1
N=1024 :  3 1 1 1 1 1 1 1  1 1 1 1 1 1 1 1
...
```

**Fig. 6.** Excerpt from the variant dispatch table $V_{\texttt{sort}}$. Each entry points to the expected fastest `sort` variant for a given call context $(n, p)$. For better readability we show here integer indices instead, where 1 denotes a call to ParQS, 2 to ParMS, 3 to SeqQS and 4 to ParIS. Here, we use a quad-logarithmic scale (with a given upper bound) on the problem size axis; lookup returns the nearest tabled entry.

mined in linear time for a parallel divide degree of 2, as in `ParQS` and `ParMS`. Fig. 6 shows an excerpt of the $V_{sort}$ table computed by the dynamic programming algorithm.

As a fully automatic composition tool interpreting metacode syntax is not yet available, we simulated the effect of composition by injecting the the dispatch tables and lookups by hand into the appropriate places in the source code. For evaluation purposes, the resulting Fork source code can be configured to either use the schedule and variants as given in the computed dispatch tables or the original component implementations without modification. The code is compiled to the SBPRAM [8] and executed on its cycle-accurate simulator, run on a SUN Solaris server.

Figure 7 shows average times for sorting 1023 numbers on up to 32 PRAM processors (left) and a section of the variant dispatch table $V_{\texttt{sort}}$ (right). We can observe that all parallel variants outperform sequential quicksort. Among the former, adaptive parallel quicksort and our performance-aware composition perform best. That these two variants perform almost equivalent comes at no surprise when looking the $V_{\texttt{sort}}$ table entries: except for small problem sizes, mostly quicksort is selected.

We observe that, for the composition of our four sorting components, the performance improvements of the composed function (up to a factor of 10 compared to sequential sorting and up to a factor of 4 and 5 compared to the parallel quicksort and mergesort only, resp.) are due to schedule lookup. The gain increases with $N$ because the general case of two recursive subtasks gets more common.

## 6 Related work

Automatic program specialization has been a great concern for many years; the cost of genericity is sometimes too big to be acceptable, hence the interest in specialization for specific applications.

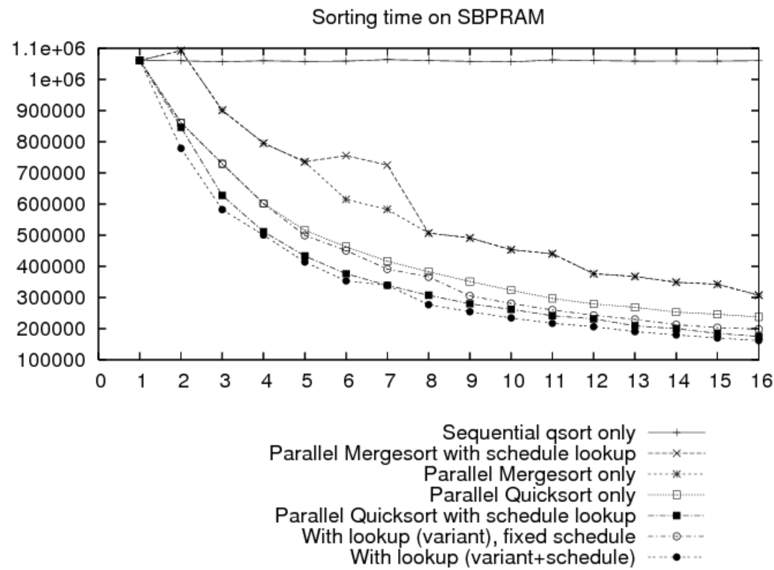



**Fig. 7.** Execution times (in SBPRAM clock cycles) of the composed sorting program, applied to a fixed problem size of $n = 1023$ numbers on up to $p = 32$ SBPRAM processors, compared to the times needed when using each component exclusively. The dispatch tables were precomputed by dynamic programming for $N = 1024$ and $P = 32$ (cf. Fig. 6).

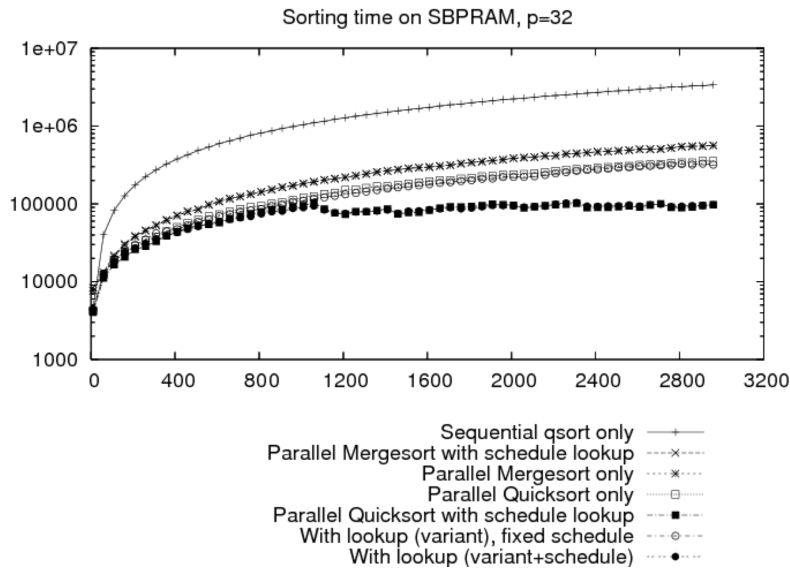



**Fig. 8.** Timing results (in SBPRAM clock cycles) for non-composed and composed sorting with a fixed number $P = 32$ of SBPRAM processors and varying problem size.

For object-oriented languages, the work of Schultz *et al.* [16] demonstrates how advices from the developer may be used to automatically specialize applications. We previously generalizes this idea by optimally composing special implementations of data structure and algorithms, using matrix multiplication as an example domain [1]. Our work may be considered as a generalization of the dispatch mechanism in object-oriented languages. Recent work on Context Oriented Programming (COP) [18] offers generic, language level, mechanisms suitable for implementing context-aware optimizations in run time.

Adaptive optimizations have recently received an increasing attention in the field of high-performance computing. One example is the optimization of cache behavior of library routines. Several approaches to algorithm selection, resource allocation, or scheduling at run-time have been proposed in literature. However, these are often for specific domains, and dynamic selection of the data representation is hardly considered. Also, no approach supports the combinations of these three subproblems, and only one considers recursive components.

Domain-specific libraries generators achieve adaptive optimizations by using profile data gathered during off-line training processes to tune key parameters, such as loop blocking factors to adapt to, e.g., cache sizes. This technique is used for both generators that target sequential and parallel libraries, for example ATLAS [19] for linear algebra computations, and SPIRAL [13] and FFTW [7] for Fast Fourier Transforms (FFT) and signal processing computations. With its concept of composition plans computed at run-time, FFTW also supports optimized composition for recursive components.

Li *et al.* [11] implement a library generator that uses dynamic tuning to adapt the library to the target machine. They use a number of machine parameters (such as cache size, the size of the input and the distribution of the input) as input to a machine learning algorithm. The machine learning algorithm is trained to pick the best algorithm for any considered scenario.

For Single Program Multiple Data (SPMD) parallel systems, which are most popular in high-performance computing and programmed using MPI or partitioned global address space languages, several approaches to dynamic algorithm selection based on predicate functions generated from training data have been proposed. Brewer [4] investigated such a system for sorting and PDE solving, and also considered limited support for the dynamic selection of array distributions on distributed shared memory machines. Yu and Rauchwerger [20] investigated dynamic algorithm selection for reductions and in the STAPL [17] library for sorting and matrix computations. In these approaches, dynamic selection is only applied for flat composition; calls within the library itself are not considered. Yu and Rauchwerger use a training phase to create a model for predicting the run-time of each of the reduction algorithms. The training phase assesses the machine profile and calibrates the parameters of the model to make the predictions correct. For each call, the calibrated run-time prediction functions are evaluated and the decision is memorized so it can be reused if the same parameter configuration should occur again. This way, the overhead of the dispatch at each call is reduced.

Olszewski and Voss [14] proposed a dynamic adaptive algorithm selection framework for divide-and-conquer sorting algorithms in a fork-join parallel setup. Their approach is divided into two phases. First, they use a dynamic programming algorithm

to select the best sequential algorithm for different problem sizes. Then, they determine the threshold problem sizes for when to submit the subproblems to a shared work queue and execute in parallel rather than to execute sequentially. This method is limited to threaded parallel divide-and-conquer algorithms that are, barring the possible parallel execution of subproblems, identical to their sequential counterparts, such as Quicksort with sequential partitioning and Mergesort with sequential merging. Such algorithms do not scale well to larger numbers of processors.

PetaBricks [2], developed independently from our work [9], applies a similar approach where the variant choice functions for recursive components (cf. our tables) are not computed by dynamic programming but by a genetic algorithm, essentially applying heuristic cuts to the optimization space. Schedules and resource allocation are not co-optimized with variant selection but delegated to a run-time system with work-stealing dynamic scheduler. Numerical accuracy is considered as an additional algorithmic property in composition.

Various static scheduling frameworks for malleable parallel tasks and task graphs of modular SPMD computations with parallel composition have been considered in the literature [15,21,22]. Most of them require a formal, machine-independent specification of the algorithm that allows prediction of execution time by abstract interpretation. In our work, we separated the actual implementation from the model of its execution time. Scheduling methods for distributed memory systems also need to optimize communication for data redistribution at module boundaries. This can be added to our framework as well. To the best of our knowledge, none of them considers the automatic composition of different algorithm variants at deployment time and their automatic selection at runtime.

As we do not consider heterogeneous or distributed systems here, additional interoperability support by (parallel) CORBA-like mechanisms is not required. However, such an extension would be orthogonal to our approach.

Our earlier work explores the parallel composition operator for dynamic local load balancing in irregular parallel divide-and-conquer SPMD computations [6]. We balanced the trade-off between group splitting for parallel execution of subtasks and serialization, computing—off-line by dynamic programming—tables of the expected values of task size ratios, indexed by $n$ and $p$, where scheduling should switch between group splitting and serialization. Our schedule lookup table above can be seen as a generalization of this.

## 7 Conclusions and Future Work

The paper proposes a composition framework for SPMD parallel components. Components are specified independently of the specific runtime environment. They are equipped with meta-code allowing to derive their performance in a particular runtime (hardware) environment at deployment time. Based on this information, a composition tool automatically approximates optimal partial schedules for the different component variants and processor and problem sizes and injects dynamic composition code. Whenever the component is called at runtime, the implementation variant actually executed is selected dynamically, based on the actual problem size and the number of processors available

for this component. Experiments with two parallel and one sequential sorting component prototypically demonstrate the speed-up compared to statically composed parallel solutions.

**Static agglomeration of dynamic composition units** Coarsening the granularity of units for dynamic lookup is an optimization of our approach. We could consider the trade-off between the overhead of dynamic composition vs. the (expected) performance improvement due to choosing the (expected) fastest variant and schedule. We could consider units for dynamic composition that have a larger granularity than individual performance-aware function calls. A possible approach could be to virtually in-line composition operator "expressions" (which may span across function calls, i.e., define contiguous subtrees of the call graph) that will be treated as atomic units for dynamic composition. The composition tool would compose these units statically including a static composition of the `time` functions. This will usually somewhat decrease accuracy of predictions and miss some better choices of variants within these units but also saves some dynamic composition overhead.

**Table compression** If implemented in the naive way described above, even with logarithmic axes, the dispatch and schedule/resource allocation tables can grow very large, especially if multiple problem sizes or machine parameters are to be considered. Also, the time for computing them will grow accordingly. Hence, compression techniques need to be investigated. For instance, regions in the $V$ or $S$ tables with equal behavior could be approximated by polyhedra bounded by linear inequalities that could result in branching code instead of the table entry interpolations for dynamic dispatch and scheduling. Alternatively, machine learning techniques could be applied to automatically derive surrogate functions for the lookup mechanism from training data. Compression techniques for dispatch tables of object-oriented polymorphic calls could be investigated as well.

**Adaptation of time data parameters** In the sorting example, we used randomly generated problem instances to compute parameter tables with average execution times for `qsort`, `partition` etc. used in optimization. In certain application domains or deployment environments, other distributions of input data could be known and exploited. Moreover, expected execution times could be adjusted dynamically with new runtime data as components are executed, and in certain time intervals, a re-optimization may take place such that the dispatch tables adapt to typical workloads automatically.

**Domains of application** In scientific computing as well as non-numerical applications, there are many possible application scenarios for our framework. For instance, there is a great variation in parallel implementations of solvers for ODE systems that have equal numerical properties but different time behavior [10].

**Other parallel platforms** In the EU FP7 project PEPPHER (www.peppher.eu), started in 2010, we are applying the approach of this paper to heterogeneous and hybrid multi-/manycore systems such as Cell BE and GPGPUs with the goal of improved performance portability.

**Acknowledgments** C. Kessler acknowledges partial funding from EU (FP7 project PEPPHER, #248481), Vetenskapsrådet, SSF, Vinnova, and the CUGS graduate school.

## References


1. J. Andersson, M. Ericsson, C. Kessler, and W. Löwe. Profile-guided composition. In *Proc. of the 7th Int. Symposium on Software Composition (SC 2008)*, pages 157–164, March 2008.
2. J. Ansel et al. PetaBricks: a language and compiler for algorithmic choice. In *Proc. PLDI-2009*, *ACM SIGPLAN Not.* 44(6), June 2009.
3. Uwe Aßmann: *Invasive Software Composition*. Springer, 2003
4. E. A. Brewer. High-level optimization via automated statistical modeling. *SIGPLAN Not.*, 30(8):80–91, 1995.
5. T. Decker, T. Lücking, B. Monien: A 5/4-approximation algorithm for scheduling identical malleable tasks, *Theoretical Computer Science* 361(2):226–240, 2006
6. M. Eriksson, C. Kessler, M. Chalabine: Load Balancing of Irregular Parallel Divide-and-Conquer Algorithms in Group-SPMD Programming Environments. Proc. 8th Workshop on Parallel Systems and Algorithms (PASA'06). GI Lecture Notes in Informatics (LNI), vol. P-81, pp. 313-322, 2006
7. M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005.
8. J. Keller, C. Kessler, J. Träff: *Practical PRAM Programming.* Wiley Interscience, 2001
9. C. Kessler and W. Löwe. A framework for performance-aware composition of explicitly parallel components. In *Proc. ParCo-2007, Parallel Computing: Architectures, Algorithms and Applications, Jülich/Aachen, Germany, Sep. 2007*. IOS Press, 2008.
10. M. Korch, Th. Rauber: Optimizing locality and scalability of embedded Runge-Kutta solvers using block-based pipelining, *J. of Par. and Distr. Computing* 66:444–468, 2006
11. X. Li, M. J. Garzarán, and D. Padua. A dynamically tuned sorting library. In *Proc. Int. Symposium on Code Generation and Optimization (CGO'04)*, page 111ff. IEEE CS, 2004.
12. G. Mounie, C. Rapine, and D. Trystram: Efficient approximation algorithms for scheduling malleable tasks. Proc. 11th ACM Symposium on Parallel Algorithms and Architectures (SPAA'99), ACM Press, 1999, pp. 23–32
13. J. M. F. Moura, J. Johnson, R. W. Johnson, D. Padua, V. K. Prasanna, M. Püschel, and M. Veloso. SPIRAL: Automatic implementation of signal processing algorithms. In *High Performance Embedded Computing (HPEC)*, 2000.
14. M. Olszewski and M. Voss. Install-time system for automatic generation of optimized parallel sorting algorithms. In *Proc. PDPTA*, pages 17–23, 2004.
15. Th. Rauber, G. Rünger: Compiler Support for Task Scheduling in Hierarchical Execution Models. *J. of Systems Architecture* 45:483-503, 1998
16. U. P. Schultz, J. L. Lawall, C. Consel, and G. Muller. Towards automatic specialization of Java programs. In *Proc. 13th European Conf. on Object-Oriented Programming (ECOOP'99)*, pages 367–390. Springer, 1999.
17. N. Thomas, G. Tanase, O. Tkachyshyn, J. Perdue, N. M. Amato, and L. Rauchwerger. A framework for adaptive algorithm selection in STAPL. In *Proc. 10th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP'05)*, pages 277–288, 2005.
18. M. von Löwis, M. Denker, and O. Nierstrasz. Context-oriented programming: beyond layers. In *Proc. Int. Conf. on Dynamic Languages (ICDL'07)*, pages 143–156. ACM, 2007.
19. R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001.
20. H. Yu and L. Rauchwerger. An adaptive algorithm selection framework for reduction parallelization. *IEEE Trans. Parallel Distrib. Syst.*, 17(10):1084–1096, 2006.
21. L. Zhao, S. Jarvis, D. Spooner, G. Nudd: Predictive Performance Modeling of Parallel Component Composition, Proc. 19th IEEE Int. Parallel and Distr. Processing Symp., 2005
22. W. Zimmermann, W. Löwe: Foundations for the integration of scheduling techniques into compilers for parallel languages, *Int. J. of Comput. Science and Engineering* 1(3/4), 2005