The PEPPHER Composition Tool: Performance-Aware Dynamic Composition of Applications for GPU-based Systems

Usman Dastgeer, Lu Li and Christoph Kessler
PELAB, Department of Computer and Information Science
Linköping University, Sweden
e-mail: {usman.dastgeer, lu.li, christoph.kessler}@liu.se

***Abstract*—The PEPPHER component model defines an environment for annotation of native C/C++ based components for homogeneous and heterogeneous multicore and manycore systems, including GPU and multi-GPU based systems. For the same computational functionality, captured as a component, different sequential and explicitly parallel implementation variants using various types of execution units might be provided, together with metadata such as explicitly exposed tunable parameters. The goal is to compose an application from its components and variants such that, depending on the run-time context, the most suitable implementation variant will be chosen automatically for each invocation.**

We describe and evaluate the PEPPHER composition tool, which explores the application's components and their implementation variants, generates the necessary low-level code that interacts with the runtime system, and coordinates the native compilation and linking of the various code units to compose the overall application code. With several applications, we demonstrate how the composition tool provides a high-level programming front-end while effectively utilizing the task-based PEPPHER runtime system (StarPU) underneath.

***Index Terms*—PEPPHER project; component model; GPU-based systems; performance portability;**

I. Introduction

Heterogeneous computing has become a viable option to provide high performance while keeping energy consumption low. There is a growing trend of using graphics processing units (GPUs) and other heterogeneous chip-multiprocessors for general-purpose computations in many application domains. Besides several advantages, heterogeneity has lead to severe programmability problems on the software side. The diversity of hardware architectures of different devices (CPU, GPU etc.) in a system and associated programming models has made programming these heterogeneous systems a tedious task. Moreover, achieving close-to-peak performance on these systems require architecture-specific optimizations in the source-code which restrict portability to other architectures.

PEPPHER is a 3-year European FP7 project (2010-2012) that aims to provide a unified framework for programming and optimizing applications for heterogeneous many-core processors to enable performance portability. The PEPPHER framework [1] consists of three main parts: (1) a flexible and extensible component model for encapsulating and annotating performance critical parts of the application, (2) adaptive algorithm libraries that implement the same basic functionality across different architectures, and (3) an efficient runtime system that schedules compiled components across the available resources, using performance information provided by the components layer as well as other, execution-history-based performance information.

The PEPPHER composition tool adapts applications written using the PEPPHER component model to the runtime system. It gathers important information about the application and its components from their XML-based metadata descriptors, uses that information for static composition and generates glue code creating tasks for the PEPPHER runtime system (which is based on StarPU [2]) for dynamic composition.

This paper presents the PEPPHER composition tool. Section II describes central concepts of the PEPPHER component model. Sections III and IV describe the composition tool and its current prototype implementation. Section V presents an evaluation of the developed prototype. Some related work is listed in Section VI. Section VII concludes and suggests future work.

II. Components, interfaces, implementations

A *PEPPHER component* is an annotated software module that implements a specific functionality declared in a *PEPPHER interface*. A PEPPHER interface is defined by an *interface descriptor*, an XML document that specifies the name, parameter types and access types (read, write or both) of a function to be implemented, and which performance metrics (e.g. average case execution time) the prediction functions of component implementations must provide. Interfaces can be generic in static entities such as element types or code; genericity is resolved statically by expansion, as with C++ templates.

Applications for PEPPHER are currently assumed to be written in C/C++. Several implementation variants may implement the same functionality (as defined by a PEPPHER interface), e.g. by different algorithms or for different execution platforms. These implementation variants can exist already as part of some standard library[1] or can be provided

[1]For demonstration purpose, we have used CUBLAS [3] and CUSP [4] components for CUDA implementations as shown in Section V.

by the programmer (called *expert* programmer by Asanovic et al. [5]). Also, more component implementation variants may be generated automatically from a common source module, e.g. by special compiler transformations or by instantiating or binding tunable parameters. These variants differ by their resource requirements and performance behavior, and thereby become alternative choices for composition whenever the (interface) function is called.

In order to prepare and guide variant selection, component implementations need to expose their relevant properties explicitly to a composition tool (described later). Each implementation variant thus provides its own *component descriptor*, an XML document that contains meta-data such as:

- The provided PEPPHER interface, and required interfaces (i.e., component-provided functionality called from this component implementation).
- The source file(s) of this component implementation.
- Deployment information such as compilation commands and options.
- A reference to the *platform*, i.e. the programming model/language used for the component implementation and the target architecture. The actual platform properties are defined separately in another XML document [6]. Such platform meta-data can be used at multiple levels of the PEPPHER framework. Lookup of specific platform properties may be done by composition tool, run-time or component developers.
- Type and (min./max.) amount of resources required for execution, in terms of the target platform description's name space.
- Optionally, a reference to a (usually, programmer provided) performance prediction function that is called with a given context descriptor data structure. Prediction functions may use performance data tables determined by micro-benchmarking for the target platform.
- Tunable parameters of the component implementation, such as buffer sizes.
- Additional constraints for component selectability, e.g. parameter ranges.

The `main` module of a PEPPHER application is also annotated by its own XML descriptor, which states e.g. the target execution platform and the overall optimization goal. XML descriptors are chosen over code-annotations (e.g. pragmas) as the former are non-intrusive to the actual source code and hence provide better separation of concerns. The potential headache associated with writing descriptors in XML can be eliminated to a great extent by providing tool support, as shown later.

Composition points of PEPPHER components are restricted to calls on general-purpose execution units only. Consequently, all component implementations using hardware accelerators such as GPUs must be wrapped in CPU code containing a platform-specific call to the accelerator.

Component invocations result in *tasks* that are managed by the PEPPHER run-time system and executed non-preemptively. PEPPHER components and tasks are *state-less*. However, the parameter data that they operate on may have state. For this reason, parameters passed in and out of PEPPHER components may be wrapped in special portable, generic, STL-like *container* data structures such as `Vector` and `Matrix` with platform-specific implementations that internally keep track of, e.g., in which memory modules of the target system which parts of the data are currently located or mirrored (*smart containers*). When applicable, the container state becomes part of the call context information as it is relevant for performance prediction.

The PEPPHER framework automatically keeps track of the different implementation variants for the identified components, technically by storing their descriptors in repositories that can be explored by the composition tool. The repositories enable organization of source-code and XML annotation files in a structured manner and can help keeping files manageable even for a large project.

III. Composition tool

Composition is the selection of a specific implementation variant (i.e., callee) for a call to component-provided functionality and the allocation of resources for its execution. Composition is made *context-aware* for performance optimization if it depends on the current *call context*, which consists of selected input parameter properties (such as size) and currently available resources (such as cores or accelerators). The context parameters to be considered and optionally their *ranges* (e.g., minimum and maximum value) are declared in the PEPPHER interface descriptor. We refer to this considered subset of a call context instance's parameter and resource values shortly as a *context instance*, which is thus a tuple of concrete values for context properties that might influence callee selection.

Composition can be done either statically or dynamically. *Static composition* constructs off-line a *dispatch function* that is evaluated at runtime for a context instance to return a function pointer to the expected best implementation variant. *Dynamic composition* generates code that delegates the actual composition to a context-aware runtime system that records performance history and constructs a dispatch mechanism on-line to be used and updated as the application proceeds. Composition can even be done in multiple stages: First, static composition can narrow the set of candidates for the best implementation variant per context instance to a few ones that are registered with the context-aware runtime system that takes the final choice among these at runtime.

Dynamic composition is the default composition mechanism in PEPPHER. The *PEPPHER composition tool* deploys the components and builds an executable application. It recursively explores all interfaces and components that (may) occur in the given PEPPHER application by browsing the interfaces and components repository. It processes the set of interfaces (descriptors) bottom-up in reverse order of their components' *required interfaces* relation (lifted to the interface level). For each interface (descriptor) and its component implementations, it performs the following tasks:

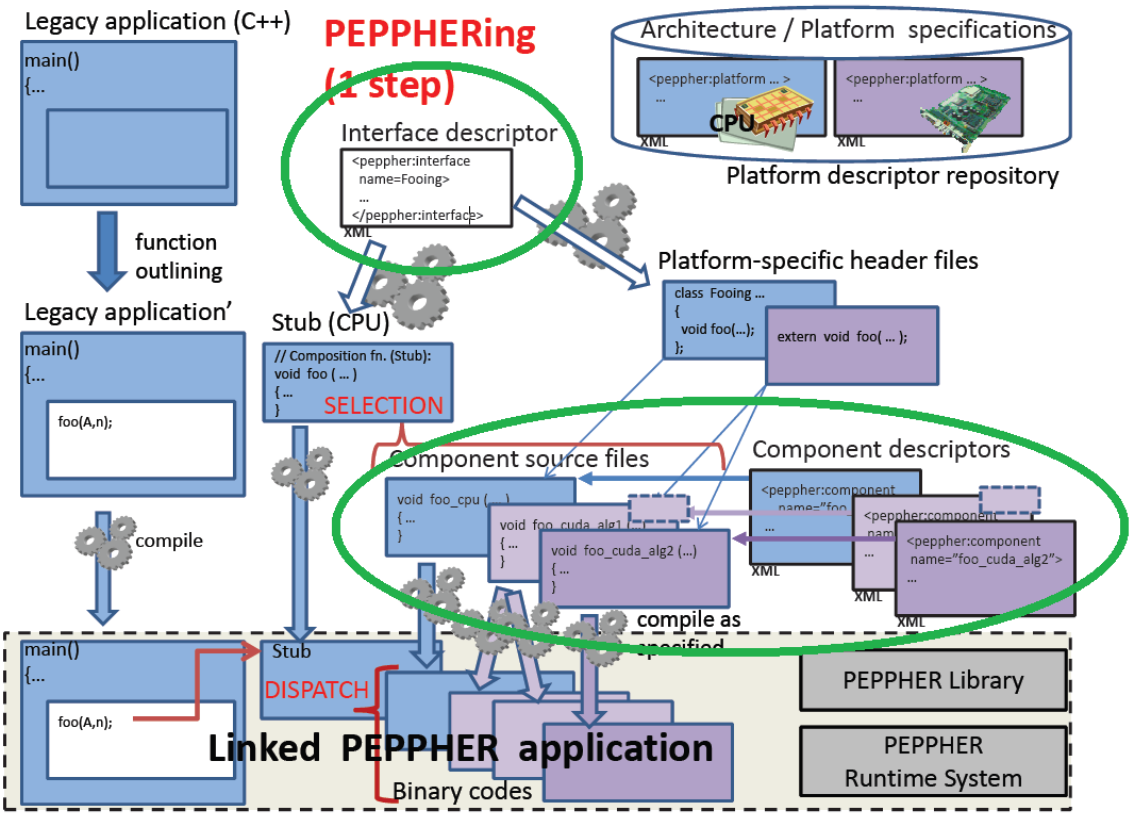


Fig. 1. Building a PEPPHER application from components using the composition tool, which coordinates all generation steps marked by the gearwheel symbols. The "PEPPHER-ization" process can be incremental, starting from a sequential legacy application running on a single CPU core, identifying components and adding implementation variants for CPU and other execution units (such as GPU) that may be available in the considered target platform(s).

1) It reads the descriptors and internally represents the metadata of all component implementations that match the target platform, expands generic interfaces and components, and generates platform-specific header files from the interface descriptor.
2) It looks up prediction data from the performance data repository or runs microbenchmarking code on the target platform, as specified in the components' performance meta-data.
3) It generates composition code in the form of *stubs* (proxy or wrapper functions) that will perform context-aware composition at runtime. If sufficient performance prediction metadata is available, it constructs performance data and dispatch tables for static composition by evaluating the performance prediction functions for selected context scenarios which could be compacted by machine learning techniques. Otherwise, the generated composition code contains calls to delegate variant selection to runtime, where the runtime system can access its recorded performance history to guide variant selection, in addition to other criteria such as operand data locality.
4) It calls the native compilers, as specified for each component, to produce a binary of every patched component source.

Finally, it links the application's main program and its compiled components together with the generated and compiled stubs, the PEPPHER library and the PEPPHER runtime system to obtain an executable program. The linking step may be architecture dependent (e.g., special handling of different executable formats may be required); the necessary command can be found in the application's `main` module descriptor.

IV. Prototype Implementation

A prototype of the composition tool has been implemented that covers the default case of dynamic composition where the run-time system selects the implementation variant. The composition tool generates low-level code to interact with the runtime system in an effective manner. Furthermore it supports component expansion for generic components written using C++ templates as well as user-guided static narrowing of the set of candidates. In this section we describe its design, main features and implementation issues.

Figure 2 shows a high-level schematic view of the prototype. Similar to typical compiler frameworks, we decouple composition processing (e.g., static composition decisions) from

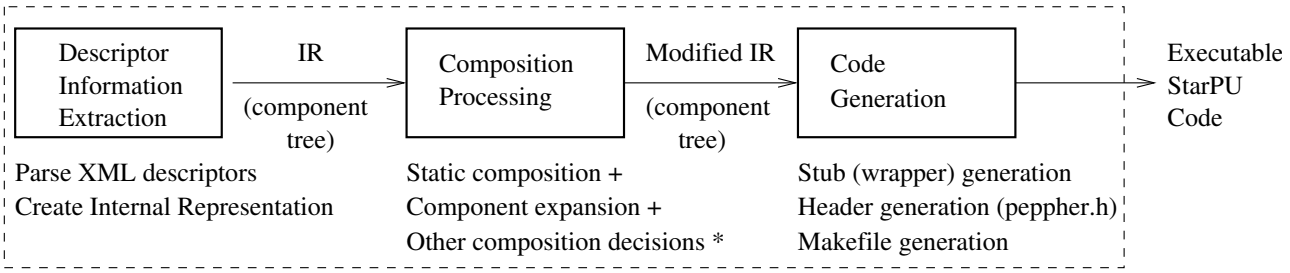


Fig. 2. Structural overview of the composition tool prototype. The features marked by + are currently only partly supported, features marked by * are not yet implemented.

the XML schema by introducing an intermediate component-tree representation (IR) of the metadata information for the processed component interfaces and implementations. The IR incorporates information not only from the XML descriptors but also information given at composition time (i.e., composition recipe). The IR can be processed for different purposes, including:

- Creating multiple concrete components from generic components by expanding template types and tunable parameters,
- Training executions to prepare for composition decisions,
- Static composition (e.g. using training executions), and
- Generating code that is executable with the PEPPHER runtime system.

A. User-guided static composition

Static composition refers to refining the composition choices at compile time, in the extreme case to one possible candidate per call and context instance. In general, static composition is supported by performance models and dispatch tables derived off-line from training runs; see [7] for details. Here, we consider the special case of *user-guided static composition*, which provides a means for the programmer to convey to the composition tool additional expert knowledge about the context that may influence the composition process. For example, a GPU implementation normally runs faster than its CPU counterpart for data parallel problems with large problem size; thus if such information is statically known, programmers may explicitly specify to compose the GPU implementation, and the overhead of the dynamic composition and the risk of wrong dynamic selection can be removed. The composition tool provides simple switches (e.g., `disableImpls`) to enable/disable implementations at the composition time without requiring any modifications in the user source-code.

B. Component expansion

Component expansion supports genericity on the component parameter types using C++ templates. This enables writing generic components such as sorting that can be used to sort different types of data. The expansion takes place statically. Component expansion for multiple values of tunable parameters to generate multiple implementation variants from a single source is not supported yet and is part of the future work.

C. Code generation

For each component interface, the composition tool generates a wrapper that intercepts each invocation of the component and implements logic to translate the call to one or more tasks for the runtime system. It also performs packing and unpacking of the call arguments to the PEPPHER runtime task handler. The directory structure provides one directory for the main component of the application and one directory for each component used. The different available implementations of a component are organized by platform type (e.g., CPU/OpenMP, CUDA, OpenCL) in different subdirectories. A global registry of interfaces, implementations and platforms helps the composition tool to navigate this structure and locate the necessary files automatically. Specifically, the tool generates:

- wrapper (stub) files providing wrapper functions for different components. Currently, one wrapper file is generated per *component*, containing one *entry-wrapper* and multiple *backend-wrappers*. The *entry-wrapper* for a component intercepts the component invocation call and implements logic to translate that component call to one or more tasks in the runtime system. A task execution can either be *synchronous* where the calling thread blocks until the task completion or *asynchronous* where the control resumes on the calling thread without waiting for the task completion. The entry-wrapper also performs packing and unpacking of arguments of a component call to the PEPPHER runtime task handler. One *backend-wrapper* for a component is generated for each backend (i.e. CPU/OpenMP, CUDA, OpenCL) for which a component implementation exists. A backend-wrapper implements the function signature

  ```
  void <name>(void *buffers[], void *arg)
  ```

  that the runtime system expects for a task function[2], and internally delegates the call to the actual component implementation which could have a different function signature.
- A header file (`peppher.h`) which internally includes all wrapper files and also contains certain other helping code

[2]As the PEPPHER runtime system is C based and the C language does not permit to call functions with varying types depending on the actual task being run.

(e.g., extern declarations). The idea of this header file is to provide a single linking point between the generated code and the normal application code. The *main* program writer only needs to include this header file to enable the composition.

- Compilation code (`Makefile`) for compiling and linking the selected (composed) components to build an executable application for a given platform.

D. Usage of smart containers

A smart container can wrap operand data passed in and out of PEPPHER components while providing a high-level interface to access that data. *Smart containers* model and encapsulate the state of their payload data. Three smart containers are currently implemented: for scalar value (`Scalar`), 1D array (`Vector`) and 2D array (`Matrix`). All three containers are made generic in the element type, using C++ templates. The containers internally implement interaction with the data management API of the PEPPHER runtime system while ensuring data consistency for data that can be accessed both by the runtime system and the application program in an arbitrary fashion. More precisely, these containers allow multiple copies of the same data on different memory units (CPU, GPU memory) at a certain time while ensuring consistency. For example, for a vector object that was last changed by a component call executed on a GPU, the vector data is copied back implicitly from GPU memory (i.e. enforce consistency) only when data is actually accessed in the application program (e.g. detected using the `[] operator` for vector objects). The containers are made portable and function as regular C++ containers outside the PEPPHER context.

Note that the composition tool also supports parameters of any other C/C++ datatypes (including user-defined datatypes) for components. However, for parameters passed using normal C/C++ datatypes, the composition tool cannot reason about their access patterns in the application program (due to pointer-aliasing and other issues) and hence ensures data consistency by always copying data back to the main memory before returning control back from the component call. Although ensuring consistency, it may prove sub-optimal as data locality cannot be exploited for such parameters across multiple component calls.

E. Inter-component parallelism

In the PEPPHER framework, a major source of parallelism comes from exploitation of independence between different component invocations. This can be obtained when a component invocation is made asynchronous so that subsequent component invocations can overlap in the actual execution. By using the smart containers, the independence between different asynchronous component invocations is implicitly inferred by the PEPPHER runtime system based on data dependencies [2].

Figure 3 depicts how usage of smart containers can help in achieving inter-component parallelism. The figure shows a simple scenario with four component calls and one vector operand on a system containing 1 CPU and 1 CUDA GPU. When the vector container `v0` is created, the payload data is placed in the main memory (*master copy*). Subsequently, depending upon the component calls using that data along their respective data access pattern (read, readwrite or write), other (partial) copies of operand data may get created in different memory units. In this case, we have CUDA device memory which has a separate address space than main memory. Assuming that all component calls are actually executed on the GPU, the figure also shows the effect of each instruction execution on the vector data state, i.e., creation/update/invalidation of data copies. As we can see, a PEPPHER container not only keeps track of data copies on different memory units but also helps in minimising the data communication between different memory units by delaying the communication until it becomes necessary. In this case, only 2 copy operations of data are made in the shown program execution instead of 7 copy operations which are required if one considers each component call independently, as done in e.g., by Kicherer et al. [8], [9], copying data each time back and forth to/from GPU device memory.

The first component call (line 4) only writes the data and hence no copy is made. Instead, just a memory allocation is made in the device memory where data is written by the component call. After the completion of the component call (line 4), the master copy in the main memory is marked outdated which means that any data access to this copy, in future, would first require update of this copy with the contents of latest copy. The next statement (line 6) is actually a read data access[3] from main memory. As the master copy was earlier marked outdated, a copy from device memory to main memory is implicitly invoked before the actual data access takes place. This is managed by the container in a transparent and consistent manner without requiring any user intervention. The copy in the device memory remains valid as the master copy is only read. Next, we have a component call (line 8) that both reads and modifies the existing data. As we assume execution of all component calls on the GPU in this scenario, the up-to-date copy already present in the device memory is read and modified. The master copy again becomes outdated. Afterwards, we have two component calls (line 10 and 12) that both only read the data. Executing these operations on the GPU means that no copy operation is required before or after the component call. Finally the statement in line 14 modifies the data in main memory so data is copied back (implicitly) from the device memory to the main memory before the actual operation takes place. Afterwards, the copy in the device memory is marked outdated and can be de-allocated by the runtime system if it runs short of memory space on the device unit. Doing so would however, require re-allocation of memory for future usage.

As all four component calls are asynchronous, the inter component parallelism is automatically inferred by the runtime system based on data dependencies. In this case, there exists a

[3]The read and write accesses to container data are distinguished by implementing proxy classes for element data in C++ [10].

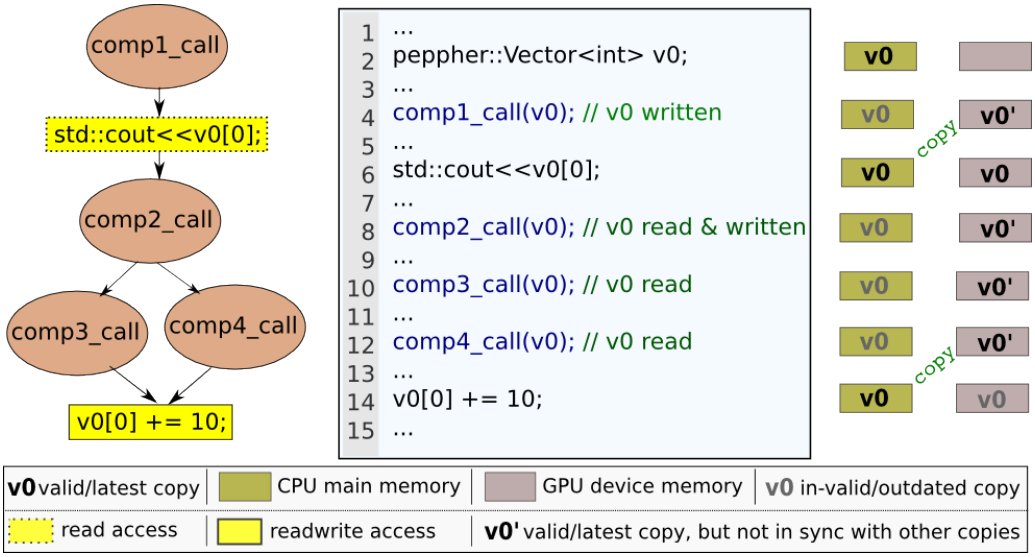


Fig. 3. Example for usage of PEPPHER smart containers with component calls and other user code. The middle part shows a code-scenario with four component calls and one vector operand on a system containing 1 CPU and 1 CUDA GPU. The effect of each instruction on the state of the data is also shown right for each instruction, assuming the four component calls are executed on the GPU. The left part highlights inter-component parallelism for asynchronous component executions based on data dependencies.

read-after-write dependency between first (line 4) and second (line 8) component call; however, independence exists between third (line 10) and fourth (line 12) component call as they access the same data but in a read only manner. Even in this simple scenario where we assume execution of all component calls on a GPU, this independence can still be exploited by doing concurrent kernel executions[4] on the GPU.

In the application program, the execution looks no different to the synchronous execution as data consistency is ensured by the smart containers. Blocking is implicitly established for a data access from the application program to a data that is still in use with the asynchronous component invocations made earlier (with respect to program control flow) than the current data access.

F. Intra-component parallelism

A common source of intra-component parallelism is a parallel component implementation, e.g. a CUDA implementation. However, for certain computations, more parallelism can be spawned from a single component invocation by partitioning and dividing the work into several chunks that all can be processed concurrently, possibly on different devices. This is achieved by mapping a single component invocation to multiple runtime (sub-)tasks rather than a single task. This comes in handy for data-parallel computations where the final result can be produced by just simple concatenation of intermediate output results produced by each sub-task (e.g. blocked matrix multiplication).

G. Support for performance-aware component selection

In the current prototype, the actual implementation variant selection is done using the dynamic scheduling capabilities of the PEPPHER runtime system. The actual implementation of performance-aware selection is made transparent in the prototype by providing a simple boolean flag (`useHistoryModels`). The support can be enabled/disabled both for an individual component by specifying the boolean flag in the XML descriptor of that component interface or globally for all components as a command line argument to the composition tool.

[4]Modern NVIDIA GPUs (e.g., Tesla C2050) supports this feature.

H. Efficient repetitive execution

Normally, the generated code could be sub-optimal when compared to an equivalent hand-written code. However, with the usage of smart containers and other enhancements in the current prototype, the generated code is optimized for repetitive and asynchronous executions. One major problem with code generation in our case comes with registering and unregistering operand data for usage with the runtime system as task operands. The runtime system schedules tasks at runtime, so data transfers for a task operand data are done automatically by the runtime system when needed between different processing units. With this data management capability, the runtime system optimizes data communication by exploiting data locality when scheduling a task. However, the registered data needs to be explicitly requested before accessing it in the application program to ensure data consistency. Un-registering operand data after each task invocation is often undesirable as it discards copies of the data across different memory units and thus disables any reuse of previous data-transfers for upcoming invocations. By using smart containers, we can effectively exploit data locality across different component invocations while ensuring data consistency for accesses from the application program.

I. Utility mode

Porting existing applications to PEPPHER framework (PEPPHER-ization) requires writing XML descriptors for interface and implementations as well as adding new implementation variants. To facilitate this process, the current prototype can generate a basic skeleton of these XML and C/C++ source files required for writing PEPPHER components from a simple

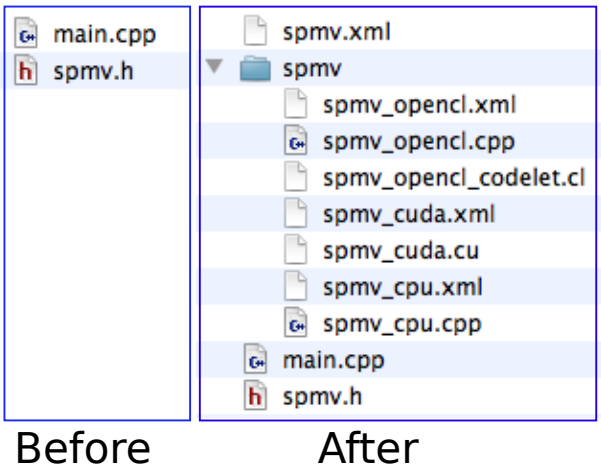


Fig. 4. Directory structure before and after generation of the basic PEPPHER component skeleton files from a simple C/C++ header file containing a method declaration. The main work left for the programmer is now to fill in the implementation details in the XML descriptor fields and provide the implementation variants' code, which can be facilitated by the existence of architecture-specific algorithms and libraries.

C/C++ method declaration. This can be really advantageous as writing XML files from scratch can become a tedious task. Our experience with porting several applications shows that the XML skeleton generation by the tool is quite useful as we are required to only fill in certain missing information. For example, the tool can successfully detect template parameters as well as suggest values for the data access pattern field of the descriptors by analyzing 'const' and 'pass by reference' semantics of the function arguments.

V. Evaluation

For evaluation, we implemented (PEPPHERized) several applications from the RODINIA benchmark suite [11], two scientific kernels (dense matrix-matrix and sparse matrix-vector multiplication) and a Runge-Kutta ODE Solver from the LibSolve library [12], using the composition tool. The main evaluation platform is a system with Intel(R) Xeon(R) CPUs E5520 running at 2.27GHz and a NVIDIA C2050 GPU with L1/L2 cache support. Furthermore, another platform with the same CPUs configuration as in the first platform but with a lower-end GPU (NVIDIA C1060 GPU) is used for performance evaluation later in this section.

A. Composition example

In the following, we will use the sparse matrix-vector multiplication to describe the complete composition (PEPPHERization) process.

The process starts with generation of basic skeleton files for components from a C/C++ header file that includes the method signature

```
void spmv(float *values, int nnz, int
          nrows, int ncols, int first,
          size_t *colidxs, size_t *rowPtr,
          float *x, float *y);
```

The composition tool can be invoked to generate component skeleton files from this method declaration: As shown in Figure 4, the command

```
compose -generateCompFiles="spmv.h"
```

generates the XML descriptors with most information prefilled as well as basic skeletons for implementation files. The programmer can then fill in the remaining details both in the XML descriptors (e.g. preferences for partitioning of input operands etc.) as well as the implementation files. For this example application, we used one serial C++ implementation for the CPU and a highly optimized CUDA algorithm provided by NVIDIA as part of their CUSP library [4]. In the application's main module `main.cpp`, we only need to add one include statement for `peppher.h` and calls to `PEPPHER_INITIALIZE()` and `PEPPHER_SHUTDOWN()` macros in the beginning and end of the `main` function, which in this case contains just a call to the `spmv` component. The last step is writing of the XML descriptor (`main.xml`) for the main function. The composition tool can now be called for generating composition code, by giving a reference to the main descriptor:

```
compose main.xml
```

This generates all the wrapper files and compilation code (makefile), necessary to compile and build the executable with the runtime support. The resulting executable can then be executed on the given platform.

TABLE I
Comparison of total source LOC (Lines of Code) written by the programmer when using the composition tool compared to an equivalent code written directly using the runtime system.

Application	Tool (LOC)	Direct (LOC)	Difference (LOC, %)
SpMV	293	376	83, 29
SGEMM	140	229	89, 63
bfs	256	364	108, 42
cfd	200	323	123, 62
hotspot	327	447	120, 37
lud	510	586	76, 15
nw	359	449	90, 25
particlefilter	652	748	96, 15
pathfinder	186	275	89, 48
ODE Solver	800	1252	452, 57

B. Productivity evaluation

The current composition tool prototype generates low-level glue-code to use the runtime system. This essentially allows application programs to run using the runtime system without requiring the programmer to actually write code for the runtime system. In the following, we compare how much we gain in terms of programming effort with this code generation functionality.

Table I shows a simple comparison of the source code written by the programmer when doing hand-written implementations for the runtime system compared to when using

Short Name	Kind	Non-zeros
Structural	Structural	2.7M
HB	HB	219.8K
Convex	Convex QP	0.9M
Simulation	Circuit Simulation	4.6M
Network	Power Network	565K
Chemistry	Quantum Chemistry	758K

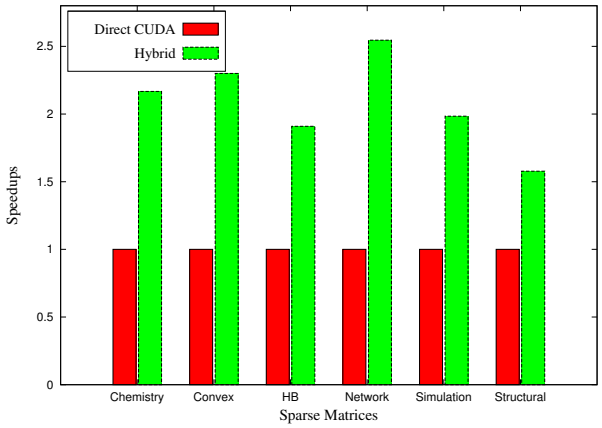


Fig. 5. Sparse matrix vector product execution for different matrices from the UF collection [14]. The performance of the PEPPHER runtime system code generated by the composition tool, shown in green (Hybrid execution), uses one CUDA GPU and all four CPUs in parallel. It is compared to a direct CUDA CUSP implementation running on the same GPU (red), where the GPU-only execution is slowed down by extensive data transfer to and from device memory, while the hybrid variant requires less communication.

the composition tool. The comparison is done using standard LOC (Lines Of Code) metric [13] for all applications. For the ODE Solver application, we have considered LOC related to the ODE solver, and not the complete LibSolve library which contains more than 12,000 LOC. As we can see in Table I, savings in terms of LOC are significant (up to 63% for `cfd` and `SGEMM`). These savings become even more significant considering the fact that the source code for the runtime system is at a low level of abstraction (plain C) and requires handling concurrency issues associated with asynchronous task executions.

The above comparison does not consider the XML descriptors which one needs to provide when using the composition tool. However, this is justified as the XML descriptor skeletons generated automatically from the C/C++ function declaration are already quite concise. For all these applications, we have used the XML descriptor generation feature of the composition tool to generate the XML descriptors and have just filled in the missing information (e.g., values for some attributes). In the remaining part of this section, we will evaluate effectiveness of the generated code by doing performance evaluation.

C. Hybrid execution

A key aspect of the PEPPHER component model is hybrid execution where execution work is distributed across all devices in the system (CPU, GPU). The advantage of hybrid execution is shown in Figure 5 where we compare hand-written CUDA code performance with the hybrid-execution capable runtime code generated automatically with our tool, using the matrices in Figure 5 from the UF collection [14] as example data. Note that the hand-written GPU execution is carried out without using our framework (marked as direct execution in Figure 5) by using the same CUDA implementation taken from the well-optimized NVIDIA CUSP library [4] that we have used for our component. The communication overhead of copying data back and forth to device memory is also included in the time measurements for GPU execution (CUDA) [15]. The measurements for hybrid execution with our framework includes overhead for data communication and task partitioning. The significant speedups with hybrid execution are due to the fact that dividing the work between CPUs and the GPU not only divides the computation but also decreases the data communication to GPU, which is a major bottleneck with GPU-only execution.

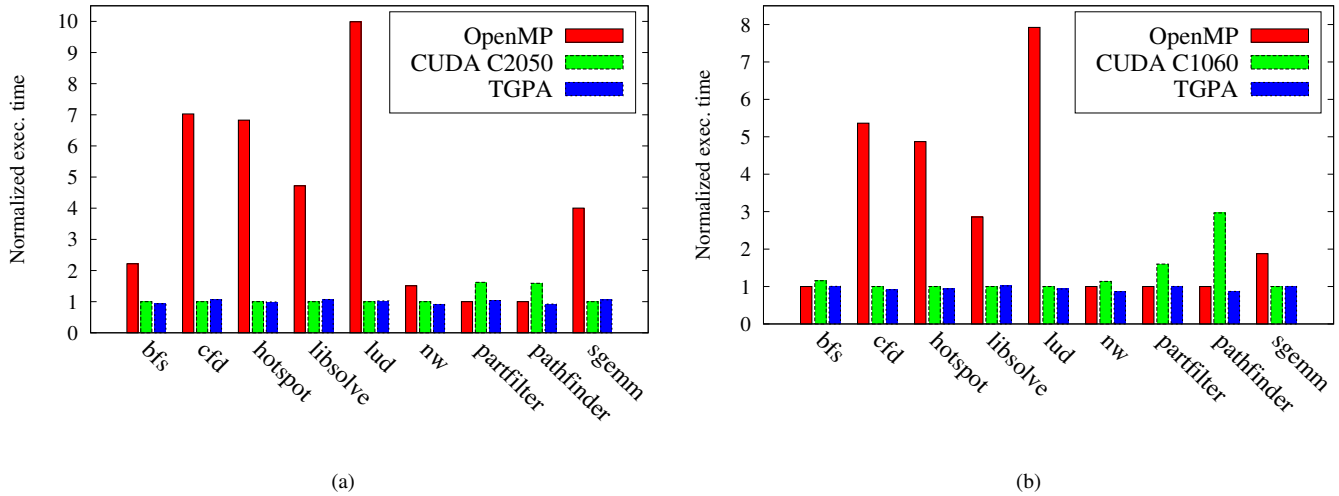
D. Dynamic Scheduling

The architectural and algorithmic differences between different devices (e.g., CPU, GPU with/without cache) and applications often have a profound effect on the achieved performance. However, these execution differences are often hard to model statically as they can originate from various sources (hardware architecture, application/algorithm, input data, problem size etc.). Dynamic scheduling can help in this case by deciding which implementation/device to use by considering previous historical execution information as well as information about current load balance, data locality and potential data-transfer cost (performance-aware dynamic scheduling).

Our tool generates necessary code for using performance-aware scheduling policies offered in the runtime system. Figure 6 shows how the usage of dynamic scheduling can help in achieving better performance on two heterogeneous architectures for a variety of applications. The execution time is averaged over different problem sizes. As we can see the execution time with generated code closely follows the best implementation from OpenMP and CUDA for all these applications. In some cases, the execution with dynamic scheduling supersedes the best static selection by making appropriate decisions for each problem size. Above all, the scheduling can effectively adjust to the architectural differences as depicted in Figure 6. This is achieved by effective utilization of the performance-aware dynamic scheduling mechanism offered by the PEPPHER runtime system.

E. PEPPHER Runtime Overhead

The main runtime overhead of the PEPPHER framework is overhead of the PEPPHER runtime system. Measuring overhead of the runtime system, in general, is a nontrivial task as it depends upon the computation structure and granularity, scheduling policy, worker types as well as hardware and software architecture. Micro-benchmarking results reported in [16] show that the task overhead of the runtime system is less than two microseconds. This overhead is negligible when considering potential gains of dynamic performance-aware



(a) (b)

Fig. 6. Execution times for applications from Rodinia benchmark suite, an ODE solver and sgemm with CUDA, OpenMP and our tool-generated performance-aware code (TGPA) on two platforms.

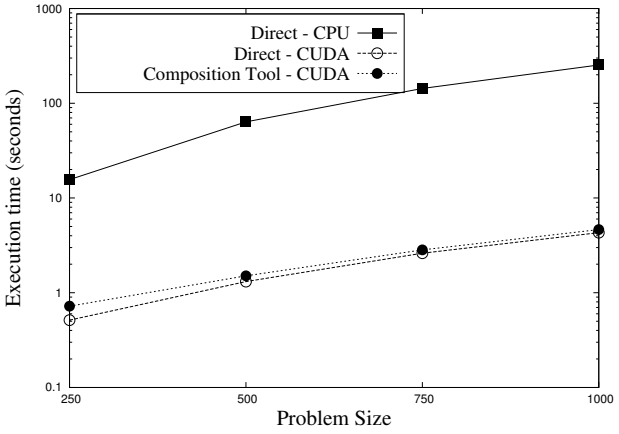


Fig. 7. Execution times for a Runge-Kutta ODE solver (libsolve) application with 9 components and 10613 invocations. Due to tight data dependency between component calls, the optimal execution results for a single powerful GPU [17]. We see that the overhead (of generated composition code for runtime task handling) compared to hand-written code is low.

scheduling and overlapping communication-computation offered by the runtime system.

Figure 7 shows the overhead of execution of the Runge-Kutta ODE solver with our framework in comparison to a direct implementation. This application is particularly interesting to measure the runtime overhead as the component calls in this application have tight data dependency which makes its execution almost sequential. The runtime overhead of the PEPPHER framework, for such a large application (10613 calls to 9 different components), is negligible in comparison to hand-written (direct) execution, as shown in the figure.

VI. Related Work

Generating stubs (wrapper or proxy functions) from formal interface descriptions in order to intercept and translate calls to components at runtime is a key concept in CORBA and subsequent component frameworks to transparently bridge the technical gaps between different languages, platforms and network locations of caller and callee, without having to change their source code. In our case, languages, platforms and network locations are equal, while the stubs encapsulate the necessary logic for determining a subset of suitable implementation variants and creating tasks for the PEPPHER runtime system. An extension to support other languages than C/C++, remote calls etc. following the CORBA approach would be a straightforward extension of our stub generation method.

Several language based systems for runtime composition of annotated implementation variants have recently been proposed in the literature, such as PetaBricks [18], Merge [19] and Elastic computing [20]. A more thorough discussion of the related work about the PEPPHER approach is presented in [1].

Our approach with performance-aware components is unique in that it does not require the programmer to use a new programming language (or a domain-specific language [19]) to implement component implementations; PEPPHER components may internally encapsulate arbitrarily specified intra-component parallelism (e.g., using pthreads, OpenMP, or arbitrary accelerator specific code) running on the resources allocated by the runtime system to the task created for its invocation, as long as the actual composition points (calls to component-provided functionality) are single-threaded, C/C++ linkable, and executed on CPUs (i.e., default execution unit with access to main memory) only. Hence, the PEPPHER component framework is, by default, non-intrusive, i.e., all metadata for components and the main program is specified externally in XML based descriptors. This enables an incremental "PEPPHER-ization" process of making legacy applications performance-portable (in principle) without modifying the existing source code.

The single-threaded call conventions, the XML format for external annotations, a non-preemptive task-based runtime system and dynamic composition as the default composition mechanism in PEPPHER are the major differences from the Kessler/Löwe components [7] which assume SPMD calls, off-line generated dispatch tables and nestable components instead.

Kicherer et al. [8], [9] proposes a C-based on-line learning solution for making dynamic selection between available implementation variants for a given invocation context. However, their approach is different as they represent each implementation variant as a dynamic library that is loaded dynamically at runtime instead of a single binary solution. Furthermore, they does not consider issue of memory management for CUDA implementation variants nor do they consider simultaneous (hybrid) computing at multiple devices present in the system.

VII. Conclusion and Future Work

Writing performance-portable applications for modern heterogeneous architectures is a non-trivial task. The component-based approach of the PEPPHER framework allows specification of multiple implementation variants for a single functionality where the expected best variant for a given execution context can be selected statically and/or dynamically. The purpose of the PEPPHER composition tool is to build applications from annotated components and thereby support "PEPPHER-izing" both new and legacy applications.

We have described the current composition tool prototype and how it could be used to deploy applications with performance-aware components on heterogeneous multi-/manycore systems. The composition tool can significantly reduce the programming complexity by transparently handling the low-level C code that interacts with the runtime system. Experiments have shown that the composition tool can successfully provide high level abstraction by automatically generating the low-level code for the runtime system. Furthermore, the generated code execute efficiently on different platforms without any need for manual tuning, thanks to careful exploitation of features provided by the runtime system.

Future versions will extend the static composition functionality with support for user-provided prediction functions [21] and enhance generic component expansion for tunable parameters. New features recently introduced in the runtime system (e.g. new types of performance models) also need to be integrated in the composition tool prototype so that they can be exploited at the component layer.

References

[1] S. Benkner, S. Pllana, J. L. Träff, P. Tsigas, U. Dolinsky, C. Augonnet, B. Bachmayer, C. Kessler, D. Moloney, and V. Osipov, "PEPPHER: Efficient and productive usage of hybrid computing systems," *IEEE Micro*, vol. 31, no. 5, pp. 28–41, 2011.

[2] C. Augonnet, S. Thibault, R. Namyst, and P. A. Wacrenier, "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures," *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009*, vol. 23, pp. 187–198, Feb. 2011. [Online]. Available: http://hal.inria.fr/inria-00550877

[3] N. corporation, "CUBLAS library: Nvidia cuda basic linear algebra subroutines." [Online]. Available: http://developer.nvidia.com/cublas/

[4] N. Bell and M. Garland, "CUSP library v0.2: Generic parallel algorithms for sparse matrix and graph computations," http://code.google.com/p/cusp-library/. [Online]. Available: http://code.google.com/p/cusp-library/

[5] K. Asanovic, R. Bodík, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. A. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. A. Yelick, "A view of the parallel computing landscape," *Communications of the ACM*, vol. 52, no. 10, pp. 56–67, 2009.

[6] M. Sandrieser, S. Benkner, and S. Pllana, "Explicit platform descriptions for heterogeneous many-core architectures," in *16th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS), International Parallel and Distributed Processing Symposium (IPDPS 2011)*, 2011.

[7] C. W. Kessler and W. Löwe, "Optimized composition of performance-aware parallel components," *Concurrency and Computation: Practice and Experience*, vol. 24, no. 5, pp. 481–498, Apr. 2012, published online in Wiley Online Library, DOI: 10.1002/cpe.1844, sep. 2011.

[8] M. Kicherer, R. Buchty, and W. Karl, "Cost-aware function migration in heterogeneous systems," in *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, ser. HiPEAC '11. NY, USA: ACM, 2011, pp. 137–145.

[9] M. Kicherer, F. Nowak, R. Buchty, and W. Karl, "Seamlessly portable applications: Managing the diversity of modern heterogeneous systems," *ACM Trans. Archit. Code Optim.*, vol. 8, no. 4, pp. 42:1–42:20, Jan. 2012. [Online]. Available: http://doi.acm.org/10.1145/2086696.2086721

[10] A. Alexandrescu, *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley, 2001.

[11] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*. IEEE, 2009, pp. 44–54.

[12] M. Korch and T. Rauber, "Optimizing locality and scalability of embedded runge–kutta solvers using block-based pipelining," *J. Parallel Distrib. Comput.*, vol. 66, no. 3, pp. 444–468, Mar. 2006. [Online]. Available: http://dx.doi.org/10.1016/j.jpdc.2005.09.003

[13] R. Park, "Software Size Measurement: A Framework for Counting Source Statements," Software Engineering Institute, Carnegie Mellon University, Tech. Rep., 1992. [Online]. Available: http://www.sei.cmu.edu/library/abstracts/reports/92tr020.cfm

[14] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. on Math. Softw. (to appear)*, 2011, http://www.cise.ufl.edu/research/sparse/matrices.

[15] C. Gregg and K. Hazelwood, "Where is the data? Why you cannot debate CPU vs. GPU performance without the answer," in *IEEE Int. Symposium on Performance Analysis of Systems and Software (ISPASS'11)*, Apr. 2011, pp. 134–144.

[16] C. Augonnet, "Scheduling Tasks over Multicore machines enhanced with Accelerators: a Runtime Systems Perspective," PhD Thesis, Universite Bordeaux 1, 2011.

[17] U. Dastgeer, C. Kessler, and S. Thibault, "Flexible runtime support for efficient skeleton programming," in *Advances in Parallel Computing, vol. 22: Applications, Tools and Techniques on the Road to Exascale Computing*. IOS Press, 2012, pp. 159–166, proc. ParCo conference, Ghent, Belgium, Sep. 2011.

[18] J. Ansel, C. P. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. P. Amarasinghe, "PetaBricks: A language and compiler for algorithmic choice," in *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009*. ACM, 2009, pp. 38–49.

[19] H. Chafi, A. K. Sujeeth, K. J. Brown, H. Lee, A. R. Atreya, and K. Olukotun, "A domain-specific approach to heterogeneous parallelism," in *16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2011, pp. 35–46.

[20] J. R. Wernsing and G. Stitt, "Elastic computing: A framework for transparent, portable, and adaptive multi-core heterogeneous computing," in *Proceedings of the ACM SIGPLAN/SIGBED 2010 conference on Languages, compilers, and tools for embedded systems (LCTES)*. ACM, 2010, pp. 115–124.

[21] L. Li, U. Dastgeer, and C. Kessler, "Adaptive off-line tuning for optimized composition of components for heterogeneous many-core systems," in *Seventh Int. Workshop on Automatic Performance Tuning (iWAPT-2012), 17 July 2012, Kobe, Japan. To appear in: Proc. VECPAR-2012 Conference, Kobe, Japan*, Jul. 2012.