Seamlessly Portable Applications: Managing the Diversity of Modern Heterogeneous Systems

MARIO KICHERER, FABIAN NOWAK, RAINER BUCHTY, and WOLFGANG KARL,
Karlsruhe Institute of Technology, Germany

Nowadays, many possible configurations of heterogeneous systems exist, posing several new challenges to application development: different types of processing units usually require individual programming models with dedicated runtime systems and accompanying libraries. If these are absent on an end-user system, e.g. because the respective hardware is not present, an application linked against these will break. This handicaps portability of applications being developed on one system and executed on other, differently configured heterogeneous systems. Moreover, the individual profit of different processing units is normally not known in advance.

In this work, we propose a technique to effectively decouple applications from their accelerator-specific parts, respectively code. These parts are only linked on demand and thereby an application can be made portable across systems with different accelerators. As there are usually multiple hardware-specific implementations for a certain task, e.g., a CPU and a GPU version, a method is required to determine which are usable at all and which one is most suitable for execution on the current system. With our approach, application and hardware programmers can express the requirements and the abilities of the application and the hardware-specific implementations in a simplified manner. During runtime, the requirements and abilities are compared with regard to the present hardware in order to determine the usable implementations of a task. If multiple implementations are usable, an online-learning history-based selector is employed to determine the most efficient one.

We show that our approach chooses the fastest usable implementation dynamically on several systems while introducing only a negligible overhead itself. Applied to an MPI application, our mechanism enables exploitation of local accelerators on different heterogeneous hosts without preliminary knowledge or modification of the application.

Categories and Subject Descriptors: C.1.3 [**Processor Architectures**]: Other Architecture Styles—*Heterogeneous (hybrid) systems*; D.3.4 [**Programming Languages**]: Processors—*Run-time environments*

General Terms: Design, Performance

Additional Key Words and Phrases: Heterogeneity, programming models, adaptive systems

ACM Reference Format:
Kicherer, M., Nowak, F., Buchty, R., and Karl, W. 2012. Seamlessly portable applications: Managing the diversity of modern heterogeneous systems. ACM Trans. Architec. Code Optim. 8, 4, Article 42 (January 2012), 20 pages.
DOI = 10.1145/2086696.2086721 http://doi.acm.org/10.1145/2086696.2086721

1. INTRODUCTION

Over the last years, heterogeneous systems have gained increasing popularity. They consist of many different processing units ranging from dedicated application accelerators to floating-point accelerators, GPUs, or integrated heterogeneous architectures

Authors' address: M. Kicherer, F. Nowak, R. Buchty, and W. Karl, Karlsruhe Institute of Technology, P.O. Box 6980, 76049 Karlsruhe, Germany. Correspondence email: kicherer@kit.edu.

DOI 10.1145/2086696.2086721 http://doi.acm.org/10.1145/2086696.2086721

like the Cell/B.E. [Chen et al. 2007]. Typically, each vendor supplies an own, proprietary programming model and toolchain for their respective hardware.

Lately, certain unifications have been attempted, ranging from RapidMind (now included in Intel's ArBB [Ghuloum et al. 2010]) to OpenCL. However, in case of the popular OpenCL, one is currently limited to the processing units supported by the system's current OpenCL library that typically only supports the vendor's hardware, e.g. the CPU in case of Intel. It remains to be seen, too, how well the OpenCL approach maps to accelerators that are different from CPU- or GPU-alike architectures. To make things even worse, comparisons with manually optimized kernels using the designated processing unit's native language can become problematic [Weber et al. 2011; Karimi et al. 2010].

For the following, we refer to the term "kernel" as a compute-intensive part of an application, usually implemented using a certain programming model, e.g. an OpenMP parallel block. An "implementation" in this context is a function that consists of one or more kernels performing a specific task on a certain CPU or accelerator and of the corresponding management code, e.g. accelerator initialization and memory transfer. However, as a kernel constitutes the relevant part of an implementation in this work, we sometimes use the term "kernel" as synonym for "implementation" in order to improve readability.

Every hardware-specific implementation usually entails additional dependencies to the application, which endanger its wide-spread employment. If, for example, an application contains code for CUDA-enabled GPUs from NVIDIA, this application will most likely not start on a system with an AMD GPU because AMD has another software stack for GPU computing and thus, the required libraries for CUDA are not available. Even if the application programmer handles the case that no CUDA-enabled GPU is available and lets execution fallback to CPU computing, the CUDA API calls remain in the code and therefore the dependency towards the CUDA runtime library. Hence, to execute this application on the AMD system, one has to install the CUDA software stack solely to execute this application, although it will only use the CPU.

Thus, a so-called fat binary approach that includes all accelerator implementations suffers from numerous dependencies due to individual toolchains and software stacks with runtime libraries. These dependencies have to be met both on the development system and on each end-user system regardless of the availability of an accelerator because the application can neither be built nor executed without having all the corresponding libraries available. In turn, those toolchains and runtime libraries can have their own dependencies, and in the worst case these can be incompatible due to different versions. Thus, installing an application as fat binary may rapidly require severe additional administrative work for potentially superfluous implementations and runtime libraries—required for accelerators currently not available in the system. In contrast, building several variants of an application for chosen sets of processing units requires additional work for building and maintaining on the developer side. If multiple types of processing units shall be used by every variant of the application, the number of possible combinations, and thus the number of variants, increases rapidly.

Therefore, we propose to decouple applications from their accelerator-specific parts by outsourcing these parts into separate libraries. This way, the application itself has minimal dependencies, and additional implementations can be loaded on demand similar to the common plugin technique. To load such libraries transparently without human intervention, a light-weight hardware-aware runtime system, called DLS, is employed that determines which library is usable without missing dependencies and hardware.

If we consider the above application example with the CPU and CUDA implementations on a system that has no CUDA-capable GPU, the runtime system would only load

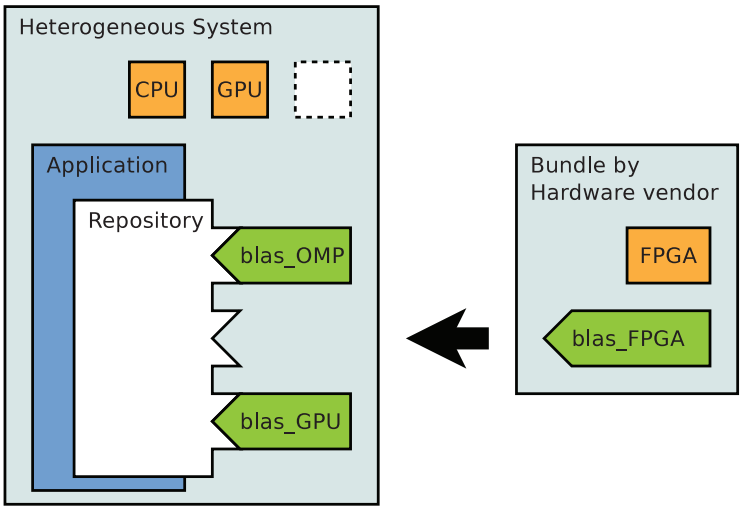


Fig. 1. Transparently extend application by a HW/SW bundle.

the CPU implementation and thus enable undisturbed execution. However, on a system with a CUDA-capable GPU, the runtime system would load both implementations. To run efficiently, we have to know which processing unit or implementation will deliver the best performance. Finding a suitable unit based on an analytical method is a very complex problem, as this requires to not only consider the properties of the accelerators but also the system's current state that influences their benefit significantly, e.g., contention. Therefore, we propose an online-learning empirical method that measures and afterwards predicts the runtime on the different processing units and thus enables to predict the fastest unit for further executions. Through periodic checks, this method is also able to help the application adapt to changing system states.

Besides portability, such a library-based approach also offers significant benefits regarding code reuse. By adding only small additional management information to the libraries, like implemented functionality and function signatures, libraries can even be reused between different applications in a transparent manner. If the management information of a library matches the requirements of an application, the runtime system loads the library, and the application can benefit from the corresponding accelerator without recompilation or binary modification. With such a mechanism, an application also automatically contains support for future accelerators. As depicted in Figure 1, hardware vendors may bundle their hardware device (e.g., FPGA, GPU, dedicated accelerator) with a tuned library that might not even exist at the application's compile time. Through our approach, the application can transparently benefit from the additional implementation and accelerator.

Besides the mentioned functionality and function signature, an application developer might want to make more precise demands towards an unknown potential implementation. Therefore, we introduce so-called attributes that can be used by application and hardware programmers to express requirements and abilities of the application and the implementations. For example in one application used during evaluation, attributes are used to demand a certain quality of random numbers. So, before loading a certain implementation, the runtime system compares the attributes to determine whether the implementation is suitable for the needs of the application.

We would like to state that within the scope of this paper, portability does not deal with issues resulting from different CPU instruction set architectures. To achieve such

portability, special techniques are required [Cha et al. 2010] or additional support from the operating system would be necessary, e.g. FatELF,[1] hampering portability nonetheless. With our mechanism, programmers can build applications for a wide set of processors and still exploit architecture-specific features: the regular CPU part of an application will be compiled for a wide-spread and common ISA like IA-32 or x86-64. Using our approach, highly optimized implementations, e.g., using vector instructions, can be executed on-demand if an appropriate CPU architecture is present.

The remainder of this paper is structured as follows: Section 2 outlines and discusses related work. In Section 3, we give a general overview of our approach, followed by implications on the application presented in Section 3.1 to 3.4. In Section 4, we present details about how the various libraries and implementation are managed. The required steps for finding the best implementation in the repository are described in Section 5. In Section 6, we provide evaluation results, and we conclude the presented work with an outlook in Section 7.

2. RELATED WORK

The growing usage of multi-core and coprocessor-extended systems has led to increased efforts in developing programming models to exploit the potential performance benefits. On the one hand, there are different approaches that map code to different types of accelerators, e.g. OpenCL, Twin Peaks [Gummaraju et al. 2010] or other CUDA-to-CPU solutions like Ocelot [Diamos et al. 2010]. On the other hand, different projects also try to efficiently distribute calculations in parallel over several processing units, e.g. StarPU runtime system [Augonnet et al. 2009] and Qilin [Luk et al. 2009].

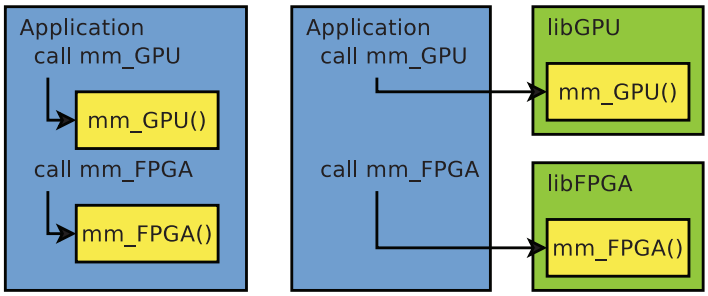
Examples for a fat binary approach are Apple's Universal Binaries and EXOCHI [Wang et al. 2007]. The latter consists of the Exoskeleton Sequencer (EXO) architecture and the C for Heterogeneous Integration (CHI) programming model. The EXO part integrates heterogeneous accelerators through an MIMD extension to the IA32 ISA and through a shared virtual memory concept. The CHI programming model allows to include accelerator-specific assembly and domain-specific languages in a C/C++ environment by extending the OpenMP pragma approach. Similar to our approach, the CHI compiler creates special sections in the resulting binary file. However, the compiler uses these sections to store the resulting special code blocks and hence, to create a fat binary.

A special technique to circumvent the problem of different CPU instruction set architectures is presented in Cha et al. [2010]. The authors developed a method to create special program strings, so-called gadgets, that form a chain of valid instructions on all the considered CPUs. The gadgets are chosen in a way that on every CPU they either perform pointless instructions or the instructions required to jump to specific code for the current ISA.

The approach presented in this work is also related to auto-tuning projects like Atlas [Whaley and Dongarra 1998]. Such projects automatically tune certain parameters of an implementation to better fit to a specific processing unit and therefore to reduce the execution time. However, the gain of these efforts is limited by the processing unit. By using our approach, an application can choose from multiple such tunable implementations, e.g. one for each processing unit. Hence, the application would profit from the best-tuned implementation that provides the highest performance.

Linderman et al. [2009] also see the rising problem of diverse hardware in heterogeneous systems. They briefly introduce annotations to choose a suitable implementation from an available set. However, in their solution the compiler analyzes the annotations and, from them, creates so-called dispatch wrapper functions that determine whether

[1]http://icculus.org/fatelf/.



(a) Legacy application (b) Implementations in separate libraries

Fig. 2. Separating applications from accelerator-specific implementations.

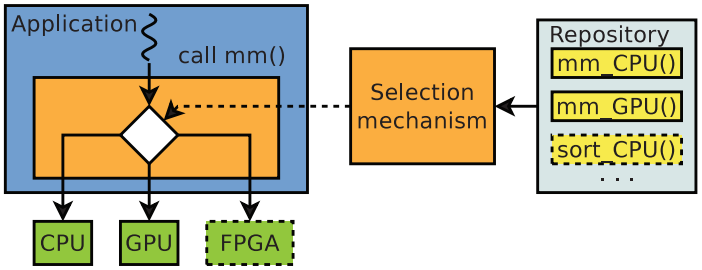


Fig. 3. Accelerator selection in a matrix multiplication example.

an implementation is suitable. In contrast, our solution is compiler-independent and allows varying amounts and values of attributes. An evaluation of their methods is completely missing. Additionally, we provide a holistic design ready to use in today's operating systems.

3. CREATING PORTABLE APPLICATIONS

One aim of this work is creating portable applications by separating the main application from accelerator-specific implementations and their dependencies. For example, instead of being part of the application binary file (Figure 2), the implementations `mm_GPU()` and `mm_FPGA()` will reside in stand-alone libraries `libGPU` and `libFPGA` (Figure 2). As a result, applications and implementations can be built and shipped individually. The details of the implications on compilation and execution are discussed in the next paragraph. As applications and implementations are fully decoupled, a mechanism is required to determine whether a kernel with certain abilities fits the requirements of an application, e.g., functionality and API compatibility. Therefore, we introduce attributes that express the requirements and abilities. Following the implications on compiling, we present our concept of attributes and their usage in more detail. Theoretically, every library on a system could contain implementations that are of value for an application. To quickly find the interesting libraries and implementations, we propose the concept of an implementation repository, discussed in the next section. At runtime of the application, this repository is queried for implementations with the required functionality. As there might be several implementations offering a certain functionality, we have to further reduce the set of possible candidates to find the best fitting one by using a special selection mechanism that is explained in Section 5.

In Figure 3, we give an overview of what happens at runtime of a thread in an application that is going to calculate a matrix multiplication. By using our concept, the thread

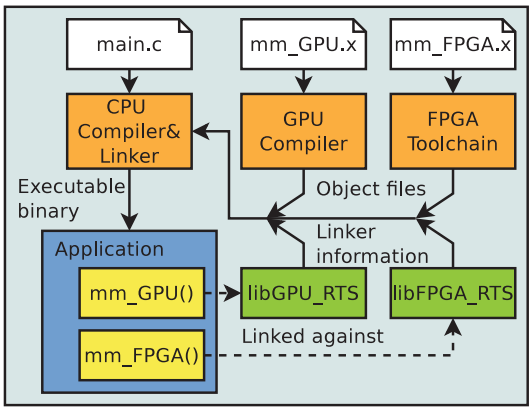


Fig. 4. Regular compilation flow—every toolchain and RTS library required to create application.

is not statically bound to a function executing on a certain processing unit. Rather, kernels calculating the product of matrices are searched. Here, `mm_CPU` and `mm_GPU` are the respective candidates in the repository that provide the required functionality. Then a selection mechanism is used to choose the best fitting implementation.

3.1. Traditional Compilation and Execution

In Figure 4, the required components for conventional compilation and execution of an application are depicted: first, the source code files for the CPU and accelerators are compiled with their respective compilers or toolchains. Then the linker creates an executable from the resulting object files and links it against the libraries of the runtime systems `libGPU_RTS` and `libFPGA_RTS` of the accelerator implementations `mm_GPU` and `mm_FPGA`. As executables are usually linked dynamically to save memory, the linker only stores a list of the library names in the executable and not the libraries themselves. At application startup, this list is used by the so-called dynamic linker/loader to determine which libraries are required. So, when the application is started, the dynamic linker/loader first looks for the required libraries, loads them all, and then starts the actual execution. However, if any of the libraries cannot be found, the dynamic linker/loader will inevitably abort the program start. For heterogeneous systems, such an abort is likely to happen. Due to the multitude of different processing units and individual programming models, it is most likely that a specific runtime library, especially in a certain version, is not available on a target system.

3.2. Decoupling Applications and Hardware-Specific Implementations

To minimize the dependencies of an application, we propose to separate the application from hardware-specific implementations. In contrast to traditional compilation in Figure 4, the resulting parts are compiled individually as depicted in Figure 5: the application is only linked against the library of our DLS runtime system and every accelerator-specific implementation is compiled independently with their respective toolchain as illustrated for GPU and FPGA, for example. Hence, application and kernels can even be developed individually on different systems, what equals the practice in many projects, where a part of the developers work on the application logic and another part is working on hardware-related optimization of compute-intensive code.

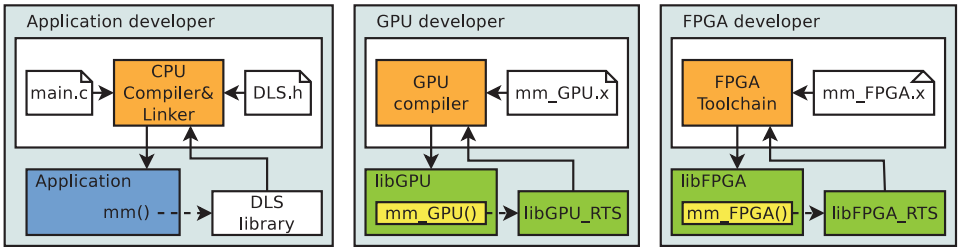


Fig. 5. Separate compilation of applications and accelerator-specific implementations.

Table I. Selection of Attributes Suitable to Express Requirements and Abilities

Name	Description	Example
`precision`	Number format	single, double, quad, GMP, exact
`signature`	Function signature	`int f(char *)`
`psize`	Problem size	100, 5000+
`target`	Type of processing unit	FPGA, GPGPU, CMP, SMP
`pmodel`	Programming model	CUDA, OpenMP, HMOL
`cpu_features`	Implementation requirements	MMX, SSE*, AVX
`speed`	Coarse performance	fast, slow

3.3. Attributes

One problem of the decoupled concept is supplying adequate information about application requirements and implementation abilities, so the DLS runtime system can determine the best implementation. For this purpose, we propose `key=value` tuples, called attributes. In Table I, we provide a selection of attributes that can be used by application and implementation developers. In the following, we discuss their individual purpose.

Certain algorithms require high precision, e.g., for numerical stability. Here, not only single-precision calculations must be prohibited, but also data-argument passing must be checked as passing single-precision data to functions expecting double-precision parameters will break due to the size of data in memory. For expressing precision requirements, we introduce the according `precision` attribute. For data passing, this attribute is sufficient if the function signature only differs in the types of floating point variables. To ensure API compatibility in general, a `signature` attribute can be used to assure that the function arguments and return values match exactly. Hence, applications can use previously unknown libraries of other applications and profit from a new accelerator. In the given example, we used the C language to define the function signature. As call conventions can differ between different languages and compilers, a comprehensive formal specification is required that defines the types and correct order of the function parameters. However, developing such a specification is beyond the scope of this paper and left aside for future work.

The benefit of implementations compared to others can vastly differ for varying problem sizes. This is especially true in heterogeneous systems. In many cases, accelerators require explicit data transfers because they come with their own dedicated memory. So, their usual performance benefit for kernel calculations has to compensate the overhead created by memory transfer. Other factors are, for example, initialization costs, like kernel compilation and transfer in case of GPUs. Hence, in most cases, using an accelerator is only beneficial if the problem size, e.g., the size of two matrices, exceeds a certain level. Again, the hardware developer can make presumptions about the implementation. For example, the time required for kernel initialization and memory transfer in a GPU case can easily take more than several milliseconds. In such a case,

the probability that the GPU is faster than the CPU for small problem sizes is rather low. So, the programmer can annotate the implementation with the `psize` attribute to e.g. express a safe lower bound for the minimum problem size from where a GPU could *possibly* be beneficial. With such information, the overhead of the history-based selector can be greatly reduced, as evaluating slow implementations can be avoided.

For organizational purposes, one can use the `target` or `pmodel` attributes to manually specify the processing unit type or the employed programming model of the implementation. As mentioned, attributes are in most cases used to express requirements of applications and abilities of implementations. However, implementations can have requirements, too. As mentioned in the introduction, kernels can make use of specific CPU instruction set extensions, e.g. SSE* instructions, and hence are dependent on the CPU model. In such a case, one can annotate an implementation with the `cpu_features` attribute and the DLS runtime system has to check the hardware before such a kernel can be used.

In certain cases, the hardware programmer can estimate the to-be-expected performance of the implementation in advance. For example, for the mentioned reference CPU implementation used as fallback, or for debug versions with verbose output, one can assume that these should be considered the last alternative for execution. To denote such implementations, we introduce the `speed` attribute with the value `slow`. On the other side, programmers can also mark implementations as `fast`. If, for example, the possible configurations of target systems are limited and known, the programmer can evaluate the implementations himself and determine which is the fastest and thus greatly reduce the time consumption for the learning process.

In the current state of DLS, the programmer is responsible for the declaration of attributes. Some attributes like the signature could be generated automatically by a compiler. Most other attributes like those presented in the evaluation highly depend on the type of computation and usually result from a performance-quality trade-off. Thus, they are difficult to determine automatically, e.g., because the compiler does not know how much accuracy is required. Therefore, we do not constrain the use of own attributes and the programmers can freely denote new attributes that fit their needs. However, special care is required as putting too much constraints on the implementation may result in an empty set of suitable implementations.

3.4. Attributes in the Source Code

To specify the attributes in the source code, there are two options: using a low-level API approach or so-called source code pragmas (`#pragma`). The benefit of the `#pragma` approach [Nowak et al. 2010] is that it allows backwards compatibility, as most major compilers simply ignore these lines if they cannot recognize them. However, if we want to integrate these attributes, we have to employ an additional source converter that processes the `#pragma`s. Another difference is how the attributes are stored. Using the low-level approach, attributes are stored as usual strings in the binary file. With the `#pragma` source converter, the programmer has the choice to simply convert the `#pragma`s into corresponding low-level API calls or to store the attributes in additional sections of the ELF binary. In ELF binaries, sections are used to separate the different parts of an application, e.g. binary code and preinitialized values. The benefit of using additional sections is that the attributes can be read and changed easily by external tools.

In Listing 1, we provide a simplified example for an application that is calling two matrix multiplication functions. To define the requirements for these calls, we add the two emphasized lines. In the first, we use the `#pragma` approach to define the required functionality `mm` and a precision attribute. In the second, a low-level API call is listed with one explicit precision attribute and the `mm` functionality denoted implicitly

Listing 1. Attributing Applications with Requirements

```
int main (int argc, char **argv) {
  double *A, *B, *C;

  /* do something */

  #pragma DLRS func=mm precision=double
  matrix_mul (A, B, C);

  /* do something */

  dls_call_attributes(mm, 1, precision, double);
  mm(A, B, C);
}
```

Listing 2. Attributing Implementations

```
#pragma DLRS func=mm target=cpu precision=double
void matrix_mul( double *A, *B, *C ) {
  /* do mm */
}

char * mm_CPU_attr = \
        dls_attributes(1, precision, double);
void mm_CPU( double *A, *B, *C ) {
  /* do mm */
}
```

by the function name. Following the same scheme, we also give an example of two implementations in Listing 2. Again, denoted explicitly and implicitly, both have the `mm` functionality, target the CPU as processing unit and calculate with double-precision values. As mentioned, the low-level approach uses implicit attributes and therefore requires a naming scheme that implicates certain attributes and is discussed in detail in Section 4.3.

4. HOST-LOCAL IMPLEMENTATION REPOSITORY

Due to several reasons, a large amount of libraries and implementations can be available on a single host: firstly, the possibly numerous different functionalities such as numerical calculations and image processing; secondly, different variants, e.g., single vs. double precision; and thirdly, for various types of processing units. To find the potentially interesting libraries and implementations, we introduce the concept of a so-called implementation repository containing detailed information about locally available implementations. In contrast to applications and implementation libraries themselves, the data in the repository, like the location in the filesystem, is host-specific and therefore, the repository has to be generated for every target system.

The implementation repository comprises a static part, stored on the hard-disk, and a dynamic part, generated at runtime. The static part actually consists of an implementation database containing the location of the available libraries, and of the performance

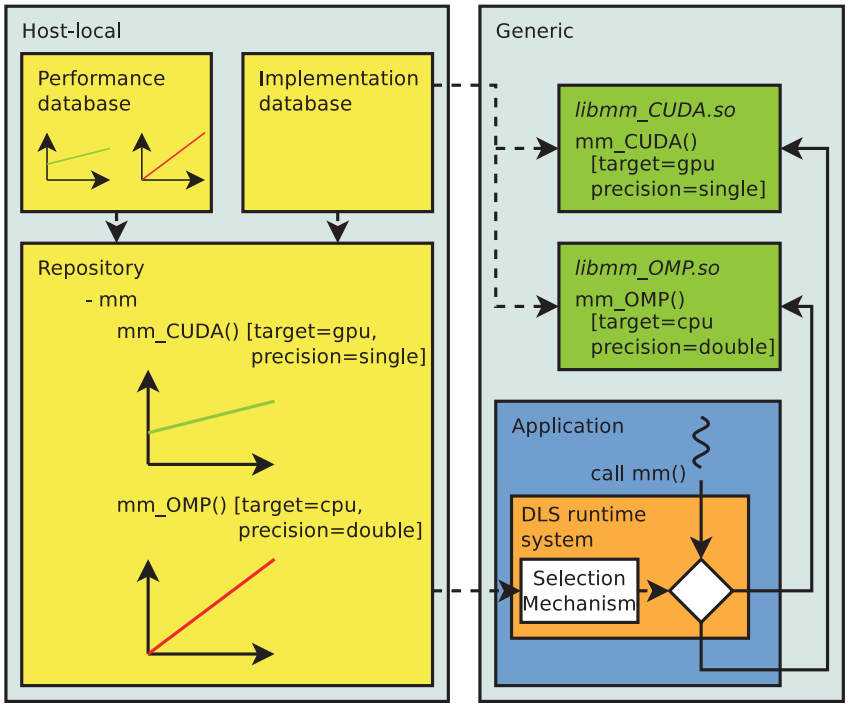


Fig. 6. Assembly of the implementation repository and interaction with the DLS runtime system.

database containing the prior time consumption of the implementations as depicted in the upper left of Figure 6. To build the complete repository at runtime of the application, the DLS runtime system, for clarity drawn inside the application, first queries the implementation database with the required functionality (here `mm`). Through this query, it receives a list of libraries that contains the respective implementation, namely `libmm_CUDA.so` and `libmm_OMP.so`. It loads all libraries and makes use of two distinct mechanisms to find the implementations in the libraries and their attributes. Having obtained the implementations and their attributes, the DLS runtime system queries the performance database for information about the performance of the single implementations. After this step, all necessary information for the implementation repository has been gathered. In the following, we explain this process in detail.

4.1. Implementation Database

Theoretically, every library available in the system could contain an implementation with a desired functionality. However, as a large amount of system libraries exist, inspecting all of these would significantly increase the effort for finding appropriate implementations. We therefore introduce the implementation database as a lookup table to quickly find the libraries of interest. An example of the implementation database is shown in Table II. The list has two columns: the first column identifies the functionality, the second delivers the path to the library that contains at least one implementation with the respective functionality. New implementations can be inserted manually into this file or through the installer or package manager. These implementations can also add an arbitrary new functionality at any time.

As well, it is possible to add dependent functionalities with different granularity. For example, an application that plays a video stream could request `decode_frame`

Table II. Example of the Implementation Database

Functionality	Path to library
mm	/usr/lib/libmm_CUDA.so
mm	/usr/lib/libblas.so
mm	/usr/lib/libmm_OMP.so
sort	/usr/lib/libsort_CPU.so
...	

Listing 3. Directory Structure of the Performance Database

```
/var/lib/perf_db/
       perf_data_mm_cpu.db
       perf_data_mm_omp.db
       perf_data_sort_cpu.db
       ...
```

functionality. For this functionality, there might be two implementations: a hardware implementation that uses a hardware decoder and a software implementation that mostly runs on the CPU and in turn uses an accelerator for suitable sub-algorithms.

4.2. Performance Database

As already mentioned, the second static part of our repository is the performance database. We employ a mechanism that evaluates the performance of implementations in relation to problem sizes during runtime and afterwards predicts the fastest one for further runs [Kicherer et al. 2011]. Instead of evaluating the performance of common implementations for every application individually, the measured values are stored in the performance database to share this information with other applications using the same implementations. As the libraries have to be portable, we do not store this data directly in the library binary file, but instead in a separate host-local database. To maintain the data in a light-weight manner, we designed the database as files in a dedicated directory. Listing 3 gives an example for such a directory. As the data for every implementation is stored in its own file, we keep time consumption for accessing and storing measured values low.

4.3. Retrieving Implementations and Attributes

In the overview in Section 3.4, we mentioned that the DLS runtime system uses two mechanisms to retrieve implementations, respectively *functions*, and attributes in a library found through the lookup in the implementation database. The first mechanism is based on a *naming convention*, the second on the special *additional ELF section* of a library created by the source converter.

As the name suggests, the *naming convention* mechanism requires that the names of the functions follow a certain scheme. In an ELF binary file, functions and strings can be found in the address space of an application using so-called symbols. These symbols are strings themselves and, when resolved, point to a certain address in memory. To get the address of a function and of the corresponding list of attributes, stored as structured string, the DLS runtime system browses the symbol table to look for symbols following the naming convention. It finds functions with a certain functionality by looking for symbols with the scheme `func_TAG`, where `func` is the required functionality and `TAG` is one of several tags stored in the DLS runtime system, e.g., the programming model or target processing unit. This way, a programmer benefits twice from wisely chosen

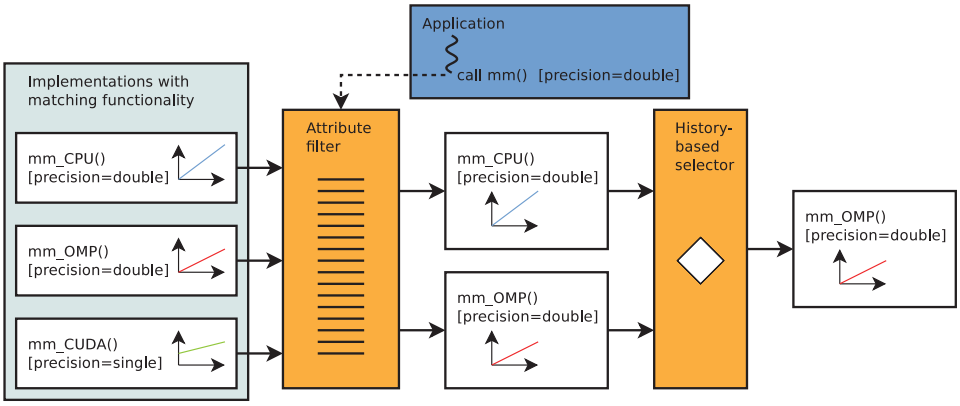


Fig. 7. Example for reduction of exploration space.

function names: they can easily identify the purpose of functions and save the effort to list these attributes, e.g. `target` and `pmodel`. Additionally, the DLS runtime system has a light-weight and standards-compliant method to retrieve the required data.

In case a function symbol following the scheme exists in the table, the DLS runtime system additionally appends the string `_attr` to the symbol. If this symbol exists, too, it points to the string containing the explicit attributes associated with the previously found function. For example, if we are looking for an implementation of a matrix multiplication (functionality `mm`), we look amongst others for the following symbols: `mm_CPU`, `mm_OpenMP`, and `mm_CUDA`. If the function shown in Listing 2 is included, the DLS runtime system finds the symbol `mm_CPU`. Next, it looks for a symbol with a further `_attr` suffix and finds the attributes referenced by the symbol `mm_CPU_attr`.

If the DLS runtime system loads a library equipped with an *additional ELF section*, it can simply read the special ELF section. The section acts as a table of contents and contains a special structure with the list of available functionalities and the address of the implementations as well as their attributes [Nowak et al. 2010].

5. REDUCING EXPLORATION SPACE USING ATTRIBUTES AND HISTORY-BASED SELECTION

Using the described mechanisms, the DLS runtime system has now a list of the locally available implementations with matching functionality and their abilities. The remaining task is to find the most suitable implementation in this set. Figure 7 gives an overview of the required steps. First, an attribute filter compares requirements with abilities, removing unsuitable implementations from the list. Here, the application demands a matrix-multiplication functionality with double precision. Three implementations for matrix multiplication exist, one in single and two in double precision. After applying the attribute filter, there are still the implementations `mm_CPU` and `mm_OMP` left. In the second step, the history-based selector determines the fastest choice by evaluating data from the performance database. In this case, the `mm_OMP` implementation promises the lowest time consumption and is therefore chosen for execution. In the following, we describe these two steps in detail.

5.1. Attribute Matching

The attribute matching is implemented as a filter: it receives selected implementations from the repository as input, removes those implementations with unsatisfying abilities, and passes the remaining ones to the history-based selector. Listing 4 shows

Listing 4. Filter Algorithm for Attribute Matching

```
foreach app->requirement as (r_key,r_value):
  foreach impl in implementation_set:
    foreach impl->ability as (a_key, a_value):
      if a_key == r_key && !match(r_value, a_value)
        remove_from_set(impl);
```

the filter algorithm in pseudo code: the algorithm iterates over all requirements of the application, then over the implementation set and over all abilities of the individual implementations therein. If the keys of the current requirement and ability are equal, it checks whether the values match. If they do not match, the implementation is removed from the set and the algorithm continues with the next implementation. However if they do match, the implementation stays in the set and is later passed to the history-based selector.

In future work, we plan to extend this matching algorithm further to allow more fine-grained control, e.g., declaring an attribute as "must match" or "may match" requirement.

5.2. History-Based Selection of the Fastest Implementation

In order to determine which implementation for a given problem size should be chosen, the selector queries the performance database where the execution times of the implementations from previous runs are stored for the respective problem size. If there are values for all implementations available, the history-based selector compares the runtimes and chooses the implementation with the shortest runtime for execution. In case not all implementations are evaluated, the selector tries to interpolate the values [Kicherer et al. 2011] using the runtimes from other problem sizes. If that is not possible, it chooses the unevaluated implementation and measures its runtime to update the database afterwards. In that way, the history-based selector predicts the fastest implementation for given problem size to be executed. Of course, this has some impact on the execution time, but as has been shown [Kicherer et al. 2011], benefit can be obtained when more accurate decisions can be made for further runs. Additionally, in certain intervals, measuring is activated nonetheless to verify the values. In case of a resource conflict, the selector can thereby adapt to new system states.

6. EVALUATION

In this section, we evaluate the benefits of our approach. In the first part, we measure the overhead of the individual components in our approach. Then we integrate our solution in the Rodinia benchmark suite in order to prove the applicability in a real-world example. To demonstrate the benefits of attributes, we prepare a use case for random number generation using different mechanisms. In the final part, we present how our approach improves the performance of an MPI application by exploiting accelerators on several hosts in parallel.

For evaluation, we use the following systems given in Table III: two GPU systems (System A and System D), a 12-core SMP system (System B), and an FPGA system (System C). Each GPU system has one NVIDIA card that is programmed using CUDA. On our FPGA system (System C), H-MOL [Kramer et al. 2009] is employed as the runtime system for the UoH HTX Board [Früning et al. 2006]. On all systems, OpenMP is used for the CPU implementation. As operating system, we use Ubuntu 10.10 x86-64, and 8.04 x86-64 for the FPGA-extended systems. If not stated otherwise, the numbers

Table III. Evaluation Systems and the Employed Programming Models and Runtime Systems

System	Type	Hardware	Progr. Model
A	CPU GPU	2× AMD Opteron Quad-Core 2378 NVIDIA GeForce GTX 275	OpenMP CUDA
B	CPU	2× Intel Xeon X5670 Six-Core	OpenMP
C	CPU FPGA	AMD Opteron Dual-Core Processor 870 Xilinx Virtex-4 FX100	OpenMP H-MOL
D	CPU GPU	Intel Core2 Quad NVIDIA GeForce 8400 GS	OpenMP CUDA

represent the average of 30 consecutive runs. For the Rodinia benchmarks, the random number generation and the MPI application, we trained the DLS runtime system in advance. As the problem size in these cases is fixed, the history-based selector requires one learning run [Kicherer et al. 2011] for every implementation before it starts predicting the fastest implementation.

6.1. Raw Overhead of the Single Components

The overhead introduced by our approach consists of a static and a variable part. The static part is caused by the additional steps required at application startup, like initialization of our mechanism and loading additional libraries. The variable part is introduced by the actual mechanisms that step in on every function call, e.g., the online-learning history-based selector.

In the first part, we evaluate the raw overhead with regard to set-up costs. For this purpose, we create a tiny application that only contains a single function executing a simple integer addition. This function is called just one time. This application takes about 3.9ms to execute. Then, we prepared the application with our concept, and now the application consumes 7.5ms. Due to its simplicity, the function call can be neglected, hence the additional one-time setup takes 3.6ms.

In order to evaluate the overhead for a single function call, we modified the application and let it execute the function 10,000,000 times, instead of one call in the previous example, to alleviate the influence of the one-time costs. Here, the original application takes about 50ms and the prepared application takes 7.8s. From this, we can calculate the per-call overhead by subtracting the costs of the single run from the overall execution time and dividing by the remaining 9,999,999 times, leading to an overhead of 775ns per function call: $\{(7.8\text{s} - 7.5\text{ms}) - (50\text{ms} - 3.9\text{ms})\}/(10{,}000{,}000 - 1) \approx 775\text{ns}$. The raw overhead introduced by our mechanism therefore is fairly low, especially when comparing it to the usual average runtime of applications and to the executed computation kernels for heterogeneous systems. Here, individual function calls may not only last several milliseconds, but can take up to several minutes, while a whole application run can last up to several hours.

We further evaluate how much the number of attributes and functions influences the time overhead for finding the best function during a call, therefore measuring the runtime with a varying number of functions and attributes. In order to get a worst-case estimation, we set up the experiment as follows: for a single run, the application's function call as well as every called function are assigned the same number of attributes. We furthermore sort the attributes in the most pessimistic way so that every attribute has to be compared and a decision cannot be made until the last attribute. Under realistic conditions, the comparisons will typically stop much earlier upon detection of first mismatching attributes, producing significantly less overhead. In Figure 8, the individual

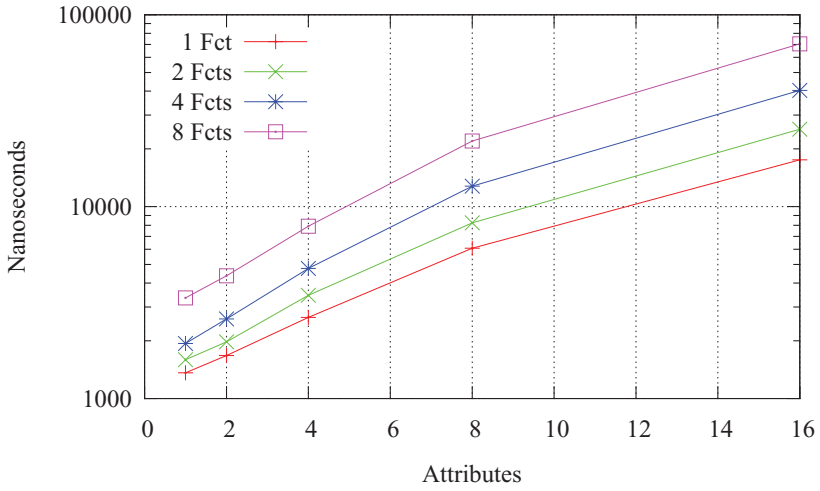


Fig. 8. Time overhead for a kernel invocation in relation to number of attributes and functions.

worst-case time consumption for different combinations is depicted. We see that the overhead depends on both the number of attributes and the number of functions.

6.2. Performance with Rodinia Benchmarks

In this experiment, we show that our mechanism introduces a negligible overhead in applications targeting heterogeneous systems and it makes these applications portable across different systems. We apply our mechanism to applications of the Rodinia benchmark suite [Che et al. 2009]. Rodinia applications have their origin among others in medical imaging, data mining and simulations. For every benchmark, an OpenMP and a CUDA version exists. To apply our concept, we just have to bundle the OpenMP and CUDA versions in separate libraries and can then create a common application calling either implementation. To assure that the executables are equal on all systems during the evaluation, we mount the benchmark directory on every system using the network file system (NFS).

In Figure 9, we see the runtimes of the applications on System A with the different approaches: our approach (called DLS) and the native versions using CUDA and OpenMP. As we can see, the application equipped with our approach has a similar runtime as the fastest one of the OpenMP and CUDA applications. This shows that the DLS runtime system successfully detects the available implementations and that the history-based selector determines the fastest one.

On System B, we start the very same applications and measure the results. As this system has no GPGPU-capable GPU, only the DLS and OpenMP results are shown in Figure 10. Although there are no CUDA GPUs and runtime libraries available in this system, all applications can still execute because they are no longer linked directly against GPGPU libraries.

6.3. Use Case: Random Number Generation

Random numbers are essential for certain tasks in cryptographic applications and simulations. In this section, we present the various ways to gain random numbers on a heterogeneous system with the Linux operating system, their differences and how a developer can use our attribute concept, so our approach will choose the best locally available mechanism for him.

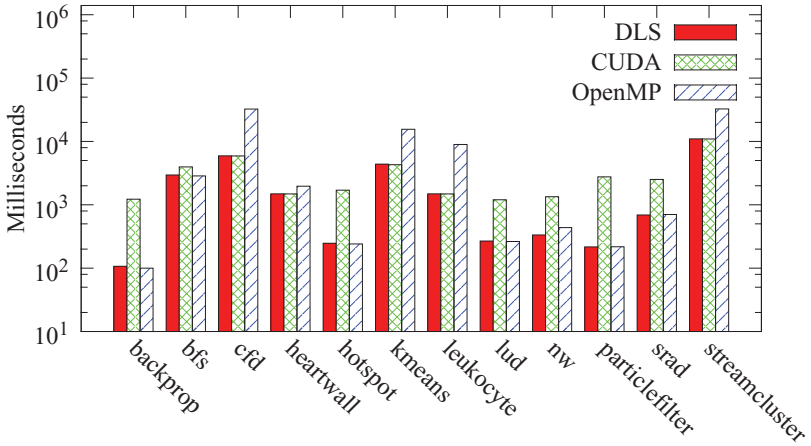


Fig. 9. Runtime of Rodinia benchmarks with our DLS, CUDA and OpenMP on GPU system A.

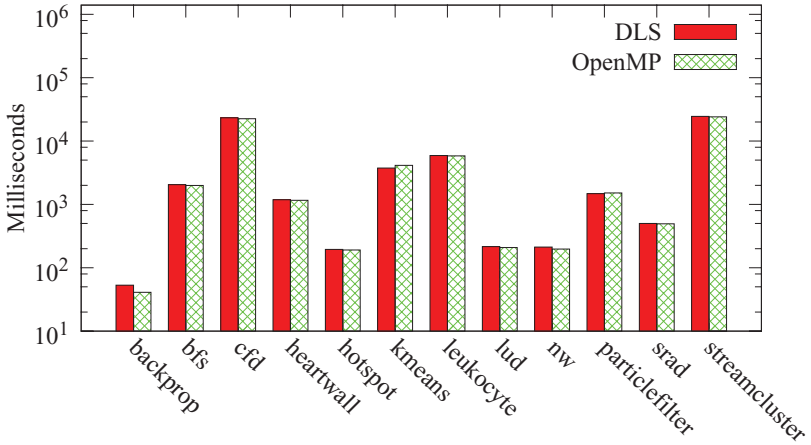


Fig. 10. Runtime of Rodinia benchmarks with our DLS and OpenMP on manycore system B.

For evaluation, we built a repository of pseudo-random number (PRN) generator libraries based on the Mersenne Twister (MT) algorithm and on the mechanisms of the Linux operating system. For the CPU, we created a Mersenne Twister implementation based on the Dynamic Creator Library.[2] An OpenMP version calls several MT random number generators in parallel. An FPGA implementation of MT is achieved by integrating the freely available hardware description[3] into our H-MOL accelerator framework for FPGAs [Kramer et al. 2009]. From the Linux system we use two methods: the GNU C library function `random()` and the kernel-based `/dev/random` virtual file. `/dev/random` delivers high-quality, *real* random numbers as long as the entropy pool of the kernel is filled. In contrast, `random()` produces only *pseudo* random numbers at a high rate. So, if a large amount of numbers is required, `/dev/random` can be unpredictably slower than `random()`. In general, `/dev/random` delivers the random numbers with the highest

[2]http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/DC/dc.html.
[3]http://www.ht-lab.com/freecores/mt32/mersenne.html.

Table IV. Attributes for Random-Number Generators

Attribute	Implementation	Value
`quality`	/dev/random MT * random()	3 (high) 2 (mid) 1 (low)
`periodicity`	/dev/random MT * random()	– 2^{19937} 2^{35}
`random_type`	/dev/random MT * random()	real pseudo pseudo
`psize`	/dev/random MT H-MOL	0-10 0-1,000

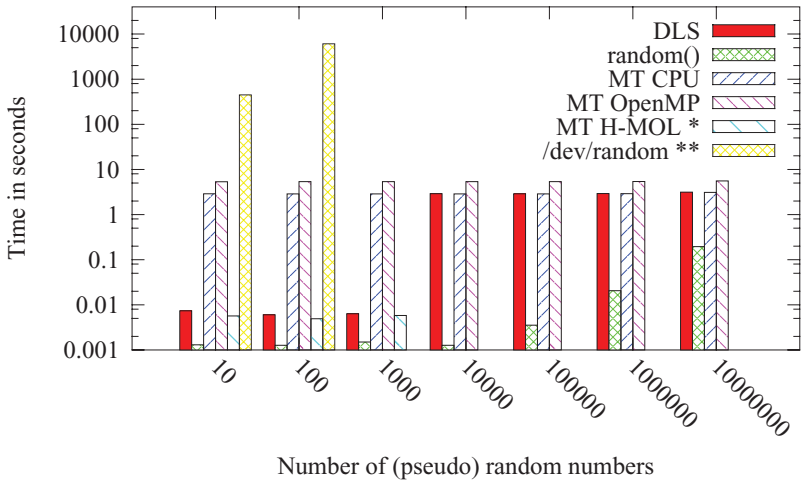


Fig. 11. Average runtime of Mersenne Twister library and built-in random-number generators on System C. (* = hardware restrictions apply; ** = 8 runs for 100)

quality. The MT algorithm provides the highest quality for pseudo random numbers and `random()` achieves the lowest quality.

To express these restrictions, we propose the attributes listed in Table IV: the first attribute, *quality*, denotes the rough quality of the random numbers.

Figure 11 illustrates our measurements of the runtime of the individual implementations on System C. Due to initialization overhead of the multiple Mersenne Twisters in the OpenMP version, speed-up is not obtained for less than 1 billion numbers. For problem sizes smaller than 1,000, our FPGA implementation (MT HMOL) performs very well because initialization is very fast. However, due to constraints of the FPGA implementation, we skip results beyond 1,000 requested numbers. For 10 and 100 requested numbers, the time consumption of the `/dev/random` implementation increases rapidly. In order to maintain clarity of the graph, we do not include results for higher amounts of numbers.

Due to the above restrictions, retrieval of random numbers poses an interesting challenge for our attribute concept. The *periodicity* attribute can be used to guarantee that the numbers are all different for a requested amount. With *random_type* it is possible to explicitly request real random numbers. With *psize*, we denote the ranges for safe and reasonable operation.

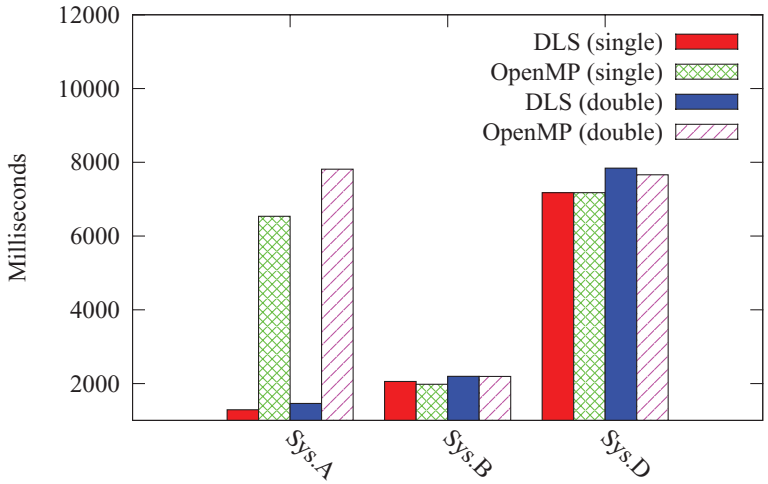


Fig. 12. Runtime of matrix multiplication example on the single MPI hosts.

We annotated the implementations as listed in Table IV and requested a good level of quality by `quality=2+`. In Figure 11, the time consumption to retrieve the respective amount of random numbers using our approach (DLS) is depicted. For 10 until 1,000 numbers, our approach selects the MT H-MOL FPGA implementation as the other MT implementations are too slow. It does not select the `random()` implementation either, although it is faster, because it produces numbers below the required quality. It does not choose the `/dev/random` implementation as well, although it provides high-quality random numbers, because it is much slower and because the quality of MT H-MOL is sufficient. Beyond 1,000 numbers, our approach chooses the MT CPU implementation, as using MT H-MOL is forbidden, MT OpenMP is too slow, and the quality of `random()` is still too low. As we see, our approach successfully chooses the most suitable from several executable implementations although the application has severe restrictions regarding the execution.

6.4. Portable MPI Application

In a further experiment, we create an application calculating a double-precision matrix multiplication using MPI. Our approach is especially interesting for such applications, as with MPI, usually multiple instances of the same application run in parallel on multiple hosts in a network. If these hosts have different heterogeneous processing units, exploiting the different host-local accelerators becomes difficult. As we will show, multiple instances of the very same MPI application will be able to benefit from such accelerators if they are using our technique.

Again, we prepare a serial, an OpenMP, and a CUDA implementation for this application. Hence, with our abstraction, the MPI application is able to take advantage of locally available accelerators on every host without recompilation or installation of superfluous runtime systems. On System D, a special obstacle is that the GPU does not support double-precision floating point numbers in hardware. Therefore, the GPU is marked with the `precision=single` attribute. To test the behavior of our solution nonetheless, we also create a single-precision variant of the application. In Figure 12, we present the individual results of both variants on the three systems A, B, and D. For better clarity of the diagram, we omitted the results of the CUDA implementation. As we can see, on System A, the application profits from the powerful GPU. On the

CPU-only System B the application can only exploit the high number of cores. However, on System D, the system with the low-performance GPU, our approach chooses the OpenMP implementation not only for double-precision but also for single-precision computation. This is caused by the low performance of the GPU, as it can not even outperform the calculation on the CPU for single precision. So, our mechanism effectively enables the parallel exploitation of different heterogeneous systems and avoids choosing an unbeneficial accelerator, which would otherwise have caused a massive slowdown.

7. CONCLUSIONS

Heterogeneous computing systems are nowadays widely accepted and in increasingly widespread use. Applications exploiting heterogeneous hardware depend on accelerator-specific libraries and therefore are not portable to different heterogeneous systems. More important, such applications benefit in different amounts from different accelerator hardware and libraries on different heterogeneous systems. To tackle the problem of creating portable applications that shall execute as fast as possible and unmodified on different systems, we propose a mechanism for decoupling applications from accelerator-specific implementations. On a target system, applications are dynamically linked with only such implementations for which an accelerator is available. This removes previously mandatory dependencies that had to be met on every target system. To increase code reuse, we introduced a runtime system that actively looks for libraries on a system that offer the required functionality and employs them transparently for the application. As several implementations of certain functionality but different properties may exist, we provide a mechanism to express requirements and abilities of applications and implementations. We employ concepts of self-organization to predict the performance of suitable implementations that meet the requirements of the application. We showed that the history-based selector is able to determine the fastest choice. Hence, our runtime system combines aspects of self-organization, i.e. runtime prediction, with dynamic adaptation to the system and its state, while both programmer and user can still influence the choice.

By evaluation we demonstrated that our mechanism introduces only negligible overhead (less than a microsecond) compared to regular execution times of applications for heterogeneous systems. We compared the time consumption of benchmarks and of an MPI application adapted to our approach against their original versions on different systems. Thereby, we could prove that our mechanism is able to dynamically select the fastest implementation on the local system. In a further experiment, we also gave an example where our attribute concept assured safe operation and a certain level of quality despite executing on different and limited hardware.

REFERENCES

AUGONNET, C., THIBAULT, S., NAMYST, R., AND WACRENIER, P.-A. 2009. StarPU: A unified platform for task scheduling on heterogeneous multicore architectures. In *Proceedings of the 15th International Euro-Par Conference on Parallel Processing (Euro-Par '09)*. Springer-Verlag, Berlin, 863–874.

CHA, S. K., PAK, B., BRUMLEY, D., AND LIPTON, R. J. 2010. Platform-independent programs. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS '10)*. ACM, New York, 547–558.

CHE, S., BOYER, M., MENG, J., TARJAN, D., SHEAFFER, J. W., LEE, S.-H., AND SKADRON, K. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*. IEEE Computer Society, Los Alamitos, CA, 44–54.

CHEN, T., RAGHAVAN, R., DALE, J. N., AND IWATA, E. 2007. Cell broadband engine architecture and its first implementation: A performance view. *IBM J. Res. Devel. 51*, 559–572.

DIAMOS, G. F., KERR, A. R., YALAMANCHILI, S., AND CLARK, N. 2010. Ocelot: A dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT '10)*. ACM, New York, 353–364.

Fröning, H., Nüssle, M., Slogsnat, D., Litz, H., and Brüning, U. 2006. The HTX-Board: A rapid prototyping station. In *Proceedings of the 3rd Annual FPGAWorld Conference*.

Ghuloum, A., Sharp, A., Clemons, N., Toit, S. D., Malladi, R., Gangadhar, M., McCool, M., and Pabst, H. 2010. Array building blocks: A flexible parallel programming model for multicore and many-core architectures. http://drdobbs.com/parallel/227300084.

Gummaraju, J., Morichetti, L., Houston, M., Sander, B., Gaster, B. R., and Zheng, B. 2010. Twin peaks: A software platform for heterogeneous computing on general-purpose and graphics processors. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT '10)*. ACM, New York, 205–216.

Karimi, K., Dickson, N. G., and Hamze, F. 2010. A performance comparison of CUDA and OpenCL. CoRR abs/1005.2581.

Kicherer, M., Buchty, R., and Karl, W. 2011. Cost-aware function migration in heterogeneous systems. In *Proceedings of the International Conference on High Performance Embeddded Architectures & Compilers (HiPEAC'11)*. ACM, New York, 137–145.

Kramer, D., Vogel, T., Buchty, R., Nowak, F., and Karl, W. 2009. A general purpose hypertransport-based application accelerator framework. In *Proceedings of the 1st International Workshop on HyperTransport Research and Applications (WHTRA'09)*. Computer Architecture Group, Institute for Computer Engineering (ZITI), University of Heidelberg, 30–38.

Linderman, M. D., Balfour, J., Meng, T. H., and Dally, W. J. 2009. Embracing heterogeneity: parallel programming for changing hardware. In *Proceedings of the 1st USENIX Conference on Hot Topics in Parallelism (HotPar'09)*. USENIX Association, Berkeley, CA, 3–3.

Luk, C.-K., Hong, S., and Kim, H. 2009. Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 42)*. ACM, New York, 45–55.

Nowak, F., Kicherer, M., Buchty, R., and Karl, W. 2010. Delivering guidance information in heterogeneous systems. In *Parallel-Algorithmen und Rechnerstrukturen*, Mitteilungen Series, vol. 27. Gesellschaft für Informatik e. V., 84–90.

Wang, P. H., Collins, J. D., Chinya, G. N., Jiang, H., Tian, X., Girkar, M., Yang, N. Y., Lueh, G.-Y., and Wang, H. 2007. EXOCHI: Architecture and programming environment for a heterogeneous multi-core multithreaded system. *SIGPLAN Not. 42,* 6, 156–166.

Weber, R., Gothandaraman, A., Hinde, R. J., and Peterson, G. D. 2011. Comparing hardware accelerators in scientific applications: A case study. *IEEE Trans. Parall. Distrib. Syst. 22*, 58–68.

Whaley, R. C. and Dongarra, J. J. 1998. Automatically tuned linear algebra software. In *Proceedings of the ACM/IEEE Conference on Supercomputing (Supercomputing '98)*. IEEE Computer Society, Los Alamitos, CA, 1–27.

Received July 2011; revised October 2011; accepted November 2011