A High-Level Compilation Toolchain for Heterogeneous Systems

W. Luk, J.G.F. Coutinho, T. Todman, Y.M. Lam, W. Osborne, K.W. Susanto, Q. Liu, W.S. Wong
Department of Computing
Imperial College London
United Kingdom
Email: {wl,jgfc,tjt97,ymlam,wgo,kws,qiangl,wswong}@doc.ic.ac.uk

ABSTRACT

This paper describes Harmonic, a toolchain that targets multiprocessor heterogeneous systems comprising different types of processing elements such as general-purposed processors (GPPs), digital signal processors (DSP), and field-programmable gate arrays (FPGAs) from a high-level C program. The main goal of Harmonic is to improve an application by partitioning and optimising each part of the program, and selecting the most appropriate processing element in the system to execute each part. The core tools include a task transformation engine, a mapping selector, a data representation optimiser, and a hardware synthesiser. We also use the C language with source-annotations as intermediate representation for the toolchain, making it easier for users to understand and to control the compilation process.

I. INTRODUCTION

A heterogeneous system contains diferent types of processing elements, ideally working harmoniously together. Depending on the application, it can include general-purpose processors (GPP), digital signal processors (DSP), and field-programmable gate arrays (FPGAs). Both embedded system-on-chip technology [1] and high-performance computing [2] can be heterogeneous. One challenge of heterogeneous computing is to improve an application by identifying and optimising parts of the program that can benefit from appropriate optimisations, and selecting the best processing element for a particular task in the program.

FPGAs are becoming increasingly popular for implementing computationally-intensive applications, providing execution speeds orders of magnitude higher than conventional processors. The key advantage of reconfigurable technology is its combination of the performance of dedicated hardware and the flexibility of software, while avoiding the cost and risk associated with circuit fabrication. The performance of a reconfigurable design is largely achieved by customising the hardware architecture to meet the application needs, and by exploiting, for instance, the inherent parallelism in the application. Furthermore, reconfigurable devices can be reused many times over for implementing different hardware architectures, thus offering more flexibility than solutions in application-specific integrated circuit technology.

The major obstacle in adopting heterogeneous systems has been the complexity involved in programming and coordinating multiple processing elements, as well as the additional effort to exploit individual specialised processing elements, which are more difficult to program than conventional processors. For FPGAs, this involves addressing the problems of hardware design, such as resource allocation and deriving efficient parallel architectures. There is a need for high-level design methods and tools that can improve designer productivity, as well as for design maintainability and portability as system requirements evolve. It is also important to facilitate design exploration so that additional target goals can be obtained, such as minimising resource utilisation and energy consumption.

In this paper, we introduce the *Harmonic* toolchain, which addresses the problem of mapping a high-level C description into a heterogeneous system. In particular, it contains four core components:

1) **Task transformation engine (Section III-C).** It allows users to select, describe and apply transformations to individual tasks, taking into account application- and platform-specific requirements. A key novelty is the use of a high-level description language, CML, which enables customisation of the transformation process to exploit features of the application domain and hardware system.
2) **Mapping selector (Section III-D).** Its aim is to optimise an application running on a heterogeneous computing system by mapping different parts of the program on to different processing elements. Our approach is unique in that we integrate mapping, clustering and scheduling in a single step using tabu search with multiple neighbourhood functions to improve the quality of the solution, as well as the speed to attain the solution.
3) **Data representation optimiser (Section III-E).** It supports optimisation of data representations, and

TABLE I: Comparison between Harmonic and related approaches. GUI and RTL stand for "Graphical User Interface" and "Register Transfer Level".

Approach	Type	Input	Task Partitioning	Source-Level Transformation	Task Mapping	Hardware Synthesis	Target
3L Diamond [3]	Commercial	C, VHDL, GUI	Manual	None	Manual	RTL	Multiprocessors
Gedae [4]	Commercial	Dataflow GUI	Manual	None	Manual	RTL	Multiprocessors, PS3
Atomium [5]	Commercial	C	Manual	Yes	Manual	Not supported	Multiprocessors
Compaan [6]	Commercial	KPN	Manual	None	Manual	Not supported	Multiprocessors
Hy-C [7]	Academic	C	Automatic	None	Automatic	Behavioural	GPP+FPGA
COSYN [8]	Academic	Task graph	Automatic	None	Automatic	Not supported	Multiprocessors
Harmonic	Academic	C	Manual/Automatic	Customisable	Manual/Automatic	RTL/Behavioural	Multiprocessors

currently targets FPGAs to allow trade-offs between computation accuracy and resource usage, speed and power consumption.

4) **Hardware synthesiser (Section III-F).** It automatically generates efficient hardware designs for certain types of computation. The novel aspect of this approach is that it captures both cycle-accurate and high-level information. This way, manual and automated optimisation transformations can be used separately or in combination, so that one can achieve the best compromise between development time and design quality.

This paper is structured as follows. Section II compares our work with existing approaches. Section III provides an overview of our design flow and its core components. Section IV illustrates our approach, and finally Section V concludes and discusses future work.

II. RELATED WORK

There are several approaches that map high-level software applications into heterogeneous systems. Table I provides a comparison between our approach, Harmonic, and other academic and commercial development tools.

The commercial tools ([3]–[6]) in this table offer powerful graphical analysis and debugging capabilities that help users make decisions with regards to the final implementation. They also provide run-time support for a number of established heterogeneous architectures. Still, these tools rely on users to make architectural decisions such as how the application is divided and mapped to different processing elements; users are also expected to have the expertise to exploit specialised processing elements such as FPGAs.

On the other hand, academic approaches such as Hy-C [7] and COSYN [8] provide a complete automatic approach for partitioning and mapping, but it is difficult for users to fine-tune the solution and to guide these optimisations, since they work as blackboxes. In contrast, Harmonic has an automatic partition and mapping process that can be constrained by users based on specific `#pragma` annotations.

Our approach employs the C language (C99) to describe not only the source application, but also the intermediate representation at every stage of the toolchain, providing three benefits:

(a) users can process legacy software code – this is useful especially if there is a large code base in C or C++, and rewriting the code into other languages could be costly;
(b) at every level of the toolchain – such as partitioning, source transformations, or mapping selection, users can understand and modify the decisions made automatically, enabling effective interaction with the toolchain;
(c) users can abstract from the specifics of programming different processing elements – there is a single high-level source language to describe the whole application, no matter how it is mapped.

In contrast, approaches like 3L Diamond [3] require users to describe algorithms in different languages and computational models to support different processing element types, such as FPGAs. This could hinder design exploration for different partition and mapping solutions, since these decisions need to be made and committed before providing an implementation.

Unlike other toolchains, Harmonic supports a source-level transformation process that can be customised by users to meet application-specific and platform-specific requirements. Transformations can be implemented in C++ and deployed as plugins using shared libraries. Alternatively, transformations can be described using CML, a language which allows users to quickly deploy, combine and parameterise transforms. An example of an application and platform domain specific transformation is generating resource-efficient floating-point designs for FPGAs, which is described in Section III-E.

To maximise performance, our toolchain is capable of deriving efficient architectures for reconfigurable hardware (FPGA) from a C description (behavioural approach). This means that unlike approaches like Gedae [4], users do not have to use RTL methodology and implement a state-machine and specify where to place registers. Moreover, unlike approaches like Hy-C which also support behavioural approach, we allow users to use RTL to fine-tune their designs, and thus they are able to interwoven both cycle-accurate (RTL) and behavioural descriptions to find the best tradeoff between design-time and the quality of the design.

III. DESIGN FLOW

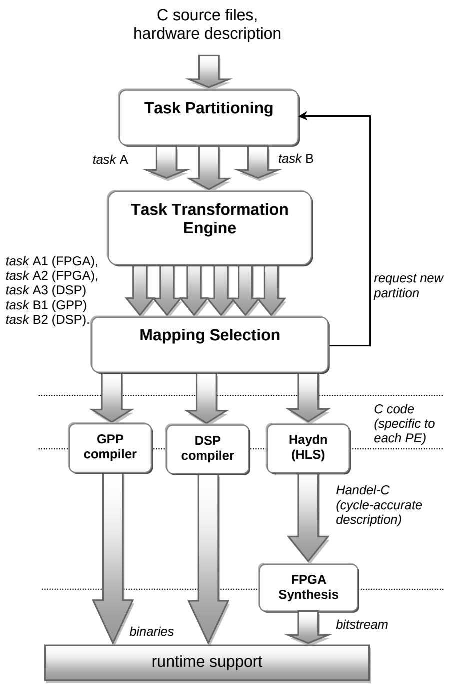
A. Overview

Fig. 1 illustrates the key components of the Harmonic toolchain. The Harmonic toolchain is built on top of the ROSE open source compiler framework [9], which provides support for source-level transformations, analysis and instrumentation of C/C++ code, and supports a number of additional frontends.

The toolchain receives as input the complete C source project as defined by a set of .c and .h source files. There are no restrictions on the syntax or semantics of the C project. By default, all source is compiled and executed on the reference processing element (usually a GPP), which serves as the baseline for performance comparison. To improve performance, Harmonic distributes parts of the program to specialised processing elements in the system.

To maximise the effectiveness of the toolchain in optimising an application, it is desirable that the C code is written in a way that it is supported by as many types of processing elements as possible in order to uncover opportunities for optimisation. For instance, recursive functions are usually limited to run on the GPP, the reference processing element, or in another instruction processor. Transforming a recursive algorithm to an iterative version involving loops can provide additional opportunities for optimisation, such as introducing a hardware pipeline running on a reconfigurable device. Hence, a set of C guideline recommendations have been defined in order to improve the quality of the mapping solutions, by making the code more migratable to different types of processing elements. We have developed a C guideline verification tool which can automatically detect many code violations of the recommendations.

The first stage of our toolchain is task partitioning, which produces a set of tasks from a C project. By default, each C function is treated as a task. A task defines the minimal computation unit that is mapped to a processing element. The goal of the task partitioner is to reduce the search space for possible mappings into the target system, by clustering two or more tasks into a single larger task (Fig. 2). From the point of the view of the task mapping process, all functions invoked inside a task are made invisible and the task is treated as a black-box. The partitioning tool also introduces OpenMP notation [10] to denote that two or more tasks can be executed in parallel because no data dependencies exist between them. Our partition approach relies on grouping functions that have similar features across different processing elements. For instance, two functions that use floating-point operations and have remarkable performance on one processing element, and mediocre performance on another, are good candidates to belong to the same cluster. Listing 1 illustrates the code annotation generated by the partition tool to indicate a cluster. Note that manual partition is possible by simply modifying the source annotations.



C source files, hardware description
Task Partitioning
task A
task B
Task Transformation Engine
task A1 (FPGA), task A2 (FPGA), task A3 (DSP) task B1 (GPP) task B2 (DSP).
request new partition
Mapping Selection
C code (specific to each PE)
GPP compiler
DSP compiler
Haydn (HLS)
Handel-C (cycle-accurate description)
FPGA Synthesis
binaries
bitstream
runtime support

Fig. 1: An overview of the Harmonic toolchain design flow.

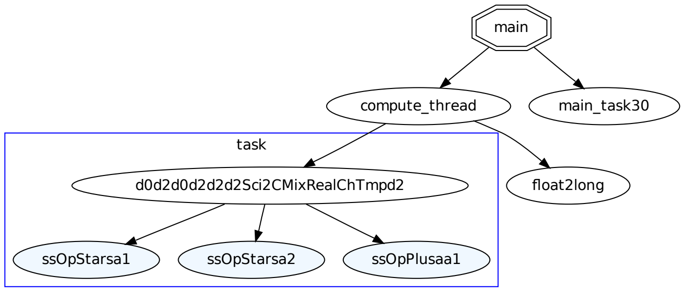


Fig. 2: The result of the task partitioning phase. By default, each individual function in the application is considered a task, and a candidate for mapping. The task partitioning tool groups small tasks into larger tasks to reduce the search space for mappings into the target system. In this example, we reduce the number of tasks from eight to five.

Once the list of tasks has been established, we use the task transformation engine to generate several C implementations for each task so that they are optimised for different processing elements. An implementation of a task can be the unrolling of a loop with an annotation to fully pipeline the outer loop, which is processed by the hardware synthesis process. Another implementation of the same task can correspond to fully unrolling the loop, but placing an annotation to share resources. Similarly, the same task can have other implementations that are optimised for DSPs and GPPs.

After the implementations have been generated, the mapping selector derives the associated cost of each implementation (either statically or dynamically), and selects the best implementation for each task so that the

```
#pragma map cluster
void d0d2d0d2d2d2Sci2CMixRealChTmpd2(...)
{
  ...
  ssOpStarsa1(a,x,t1);
  ...
  ssOpStarsa2(b,y,t2);
  ...
  ssOpPlusaa1(t1,t2,z);
}
```

Listing 1: To cluster a set of functions into a single task, we use the `#pragma map cluster` annotation as showed above on top of the function definition. In this case, all functions invoked inside the function definition are part of the same task (Fig. 2). These annotations can be introduced manually or generated automatically by the partitioning tool.

```
1  void foo(float *x, float *y) {
2     ...
3     #pragma map call_hw          \
4                 impl(MAGIC, 14)  \
5                 param(x,1000,r)  \
6                 param(h,100, rw)
7     filter(x, h);
8     ...
9  }
```

Listing 2: The mapping `#pragma` annotation (lines 3–6) is introduced by the mapping selector before each remote function call, and indicates the processing element (MAGIC DSP) and implementation (id=14) associated with it. The mapping pragma also provides information about the size of the memory referenced by the pointers, and whether this memory has been read only (`r`), written only (`w`), or both (`rw`).

overall execution time (which includes communication overhead) is minimised. Finally, the C code is generated for each processing element according to the mapping solution, and mapping `#pragmas` are introduced to indicate the association between the task and the implementation (Listing 2). If the mapping selection process is unable to meet a particular threshold, it requests the partition tool to derive a new solution. Each generated C source-file contains macros, `#pragmas` and library calls that are specific to the processing element that it was targeted for.

To derive an FPGA implementation, we use the Haydn approach [11] to perform high-level synthesis (HLS). The HLS process is guided by source-annotations which describe what hardware-specific transformations to apply, such as hardware pipelining, and capture available resources and other constraints. The output of Haydn is a Handel-C program that describes a cycle-accurate synthesisable architecture, and that can be optionally modified by the user before the hardware synthesis phase. For GPP and DSP processing elements, we use C compilers provided for those architectures. The final phase of generating and linking the binaries and providing run-time support is dependant on specific hardware platforms. In the next section we describe how different platforms can be supported by Harmonic.

B. Toolchain Customisation

The Harmonic toolchain has been built to be modular, so that new components can be introduced or combined without re-structuring the entire toolchain. In particular, each level of the toolchain parses and generates C code, which we use as our intermediate representation, along with `#pragma` annotations. Any new tool that we add only needs to follow an established convention (Listings 1 and 2).

While our approach is capable of partitioning and mapping a C project to most heterogeneous architectures, we need to customise the toolchain to exploit a specific platform. This includes providing:

1) **Platform description.** The description of the platform captures the physical attributes of the heterogeneous system in XML format, and includes information about the available processing elements, storage components, and interconnects, as well as additional information such as data types supported, storage size and bus bandwidth.
2) **Transformations.** The task transformation engine can be instructed to apply a number of optimisations to a task targeting a particular processing element (Section III-C).
3) **Harmonic drivers.** To install the backend compilation support in Harmonic, we need the necessary drivers. There are two types of drivers: *system drivers* and *processor drivers*. The toolchain expects one system driver and one processor driver for each available processing element. The system driver is responsible for determining whether a mapping solution is valid, generating code that coordinates the components of the heterogeneous system, and determining the cost of the mapping solution. The processor driver, on the other hand, works at task-level: it specifies whether a task is mappable or synthesiable for a particular processing element. It is responsible for adding any C specific idioms, and estimating the cost to execute that task.
4) **Linking and run-time support.** This module is usually provided by the hardware vendor, which includes system library calls supporting communication (DMA transfers, etc) and synchronisation calls, as well as support for low-level drivers to allow the execution of the application in a multi-processor environment.

As part of the European hArtes project, our current toolchain has been customised to target (a) the Diopsis platform [12] which contains an ARM processor and a floating-point DSP, as well as (b) the hArtes platform [13] which contains an ARM processor, a floating-point DSP, and an FPGA. An advanced partitioning tool, Zebu [14], has been interfaced to Harmonic in this customisation; Zebu is able to restructure the whole C project, for

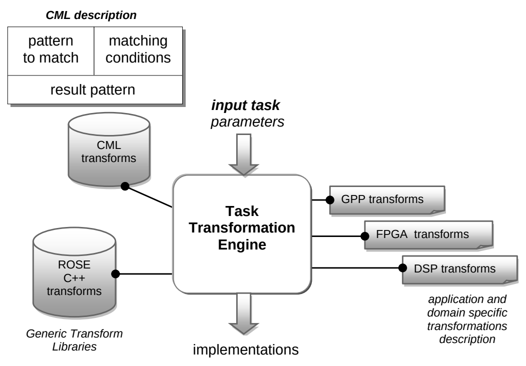


Fig. 3: An overview of the task transformation engine. The task transformation engine receives as input a task and additional parameters such as the processing element that we wish to target, and generates a set of implementations. The set of transformations to be applied to each processing element is provided by the user. The implementations of transformations are stored as shared libraries for ROSE transformations, and as text files for CML-based transformations. A CML description consists of three sections: the pattern to match, the matching conditions, and the resulting pattern (Listing 3).

instance by breaking up or merging functions, to fine-tune the granularity of the tasks according to criteria which can be complex.

The customised Harmonic toolchain is also interfaced to hArtes runtime support and compilation tools [15]. In both Diopsis and hArtes platform architectures, there is a master processing element (the ARM processor) which communicates with the remaining processing elements that act as accelerators. The C compiler for the master processor element (hgcc) is responsible for determining how data are transferred between different processing elements and what storage components to use based on the memory hierarchy. The system driver for these boards helps this process by analysing each pointer passed to a remote function, and determining the size of the memory allocated where possible, and whether the memory has been read only or written only, in order to minimise the number of memory transfers (Listing 2).

C. Task Transformation Engine

The task transformation engine (Fig. 3) applies pattern-based transformations, which involve recognising and transforming syntax or dataflow patterns of design descriptions, to source code at task level. We offer two ways of building task transformations: using the underlying compiler framework, ROSE, to write transformations in C++; this is complex but offers the full power of the ROSE infrastructure. Alternatively, our domain-specific language CML simplifies description of transformations, abstracting away housekeeping details such as keeping track of the progress of pattern matching, and storing labelled subexpressions.

CML is compiled into a C++ description; the resulting program then performs a source-to-source transformation. For design exploration, we also support interpreting

```
1   transform coalesce {
2      pattern {
3         for (var(a)=0;var(a)<expr(e1);var(a)++){
4            for (var(b)=0;var(b)<expr(e2);var(b)++)
5               stmt(s);
6         }
7      }
8      conditions {

10     }
11     result {
12        for (newvar(nv)=0;
13           newvar(nv)<expr(e1)*expr(e2);
14           newvar(nv)++)
15        {
16           var(a) = newvar(nv) / expr(1);
17           var(b) = newvar(nv) % expr(1);
18           stmt(s);
19        }
20     }
21  }
```

Listing 3: CML description of the loop coalescing transformation

CML descriptions, allowing transformations to be added without recompiling and linking. Task transformations could be written once by domain specialists or hardware experts, then used many times by non-experts. We identify several kinds of transformations: input transformations, which transform a design into a form suitable for model-based transformation; tool-specific and hardware-specific transformations, which optimise for particular synthesis tools or hardware platforms. We provide a library of useful transformations: general-purpose ones such as loop restructurings, and special-purpose ones such as transforming Handel-C arrays to RAMs.

Each CML transformation (Fig. 3) consists of three sections: 1) pattern, 2) conditions and 3) result. The pattern section specifies what syntax pattern to match and labels its parts for reference. The conditions section typically contains a list of Boolean expressions, all of which must be true for the transformation to apply. Conditions can check: a) validity, when the transformation is legal; b) applicability: users can provide additional conditions to restrict application. Finally, the result section contains a pattern that replaces the pattern specified in the pattern section, when conditions apply.

A simple example of a CML transformation is loop coalescing (Listing 3), which contracts a nest of two loops into a single loop. Loop coalescing is useful in software, to avoid loop overhead of the inner loop, and in hardware, to reduce combinatorial depth. The transformation works as follows:

- Line 1: LST 3 starts a CML description and names the transformation
- Lines 2–7: LST 3 give the pattern section, matching a loop nest. CML patterns can be ordinary C code, or labelled patterns. Here `var(a)` matches any lvalue an labels it "a". From now on, each time `var(a)` appears in the CML transform, the engine tries to match the labelled code with the source code.
- There is no conditions section, as coalescing is always valid and useful (lines 8–10: LST 3).
- Lines 11–20: LST 3 give the result section. The CML pat-

tern `newvar(nv)` creates a new variable which is guaranteed unique in the current scope. The resulting loop is equivalent to the original loop nest. The former iteration variables, `var(a)` and `var(b)` are calculated from the new variable. This allows the original loop bode, `stmt(s)` to be copied unchanged.

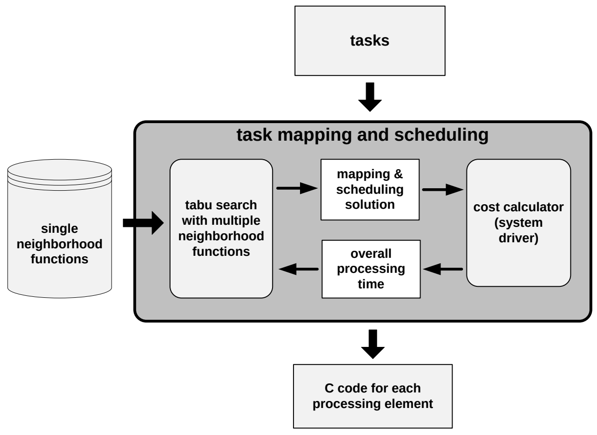
When the transformation engine is invoked, it triggers a set of transformations that are specific to each processing element, which results in a number of C implementations associated with different tasks and processor elements. The implementation description of ROSE transformations are stored as shared libraries, and the CML definitions as text files. Because a CML description is interpreted rather than compiled, users can customise the transformation by using a simple text editor, and quickly evaluate the effects of the transformation, without requiring an additional compilation stage.

D. Mapping Selection

The aim of mapping selection is to optimise an application running on a heterogeneous computing system by selecting a processing element (or more specifically an implementation) for each part of the program (Fig. 1). Our approach is unique in that we integrate mapping, clustering and scheduling in a single step using tabu search with multiple neighbourhood functions to improve the quality of the solution, as well as the speed to attain the solution [19]. In other approaches, this problem is often solved separately, i.e. a set of tasks are first mapped to each processing element, and a list scheduling technique then determines the execution order of tasks [20], which can lead to suboptimal solutions.

Figure 4 shows an overview of the mapping selection approach. Given a set of tasks and the description of the target hardware platform, the mapping selection process uses tabu search to generate different solutions iteratively. For each solution, a score is calculated and used as the quality measure to guide the search. The goal is to find a configuration with the highest score.

Fig. 5 illustrates the search process. At each point, the search process tries multiple directions (solid arrows) using different neighborhood functions in each move, which can increase the diversification and help to find better solutions. In the proposed technique, after an initial solution is generated, two neighborhood functions are used to generate neighbors simultaneously. If there exists a neighbor of lower cost than the best solution so far and it cannot be found in the tabu list, this neighbor is recorded. Otherwise a neighbor that cannot be found in the tabu list is recorded. If all the above conditions cannot be fulfilled, a solution in the tabu list with the least degree, i.e. a solution being resident in the tabu list for the longest time, is recorded. If the recorded solution has a smaller cost than the best solution so far, it is recorded as the best solution. The neighbors found are added to tabu list and solutions with the least degree are removed.



tasks
task mapping and scheduling
single neighborhood functions
tabu search with multiple neighborhood functions
mapping & scheduling solution
overall processing time
cost calculator (system driver)
C code for each processing element

Fig. 4: An overview of the mapping selection process.

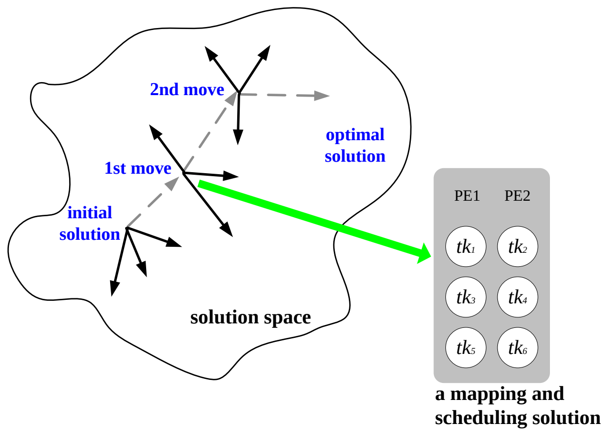


Fig. 5: Searching for the best mapping and scheduling solution using multiple neighborhood functions. The solid arrows show the moves generated by different neighborhood functions. The dotted arrows denote the best move in each iteration of the search. PE: processing element, tk: task.

This process is repeated until the search cannot find a better configuration for a given number of iterations. An advantage of using multiple neighborhood functions is that the algorithm can be parallelised, and therefore the time to find a solution can be greatly reduced.

The cost calculator (Fig. 4), which involves the Harmonic system driver (Section III-B), computes the overall processing time, which is the time for processing all the tasks using the target computing system and includes data transfer time between processing elements. The processing time of a task on a processing element is calculated as the execution time of this task on the processing element plus the time to retrieve results from all of its predecessors. The data transfer time between a task and its predecessor is assumed to be zero if they are assigned to the same processing element.

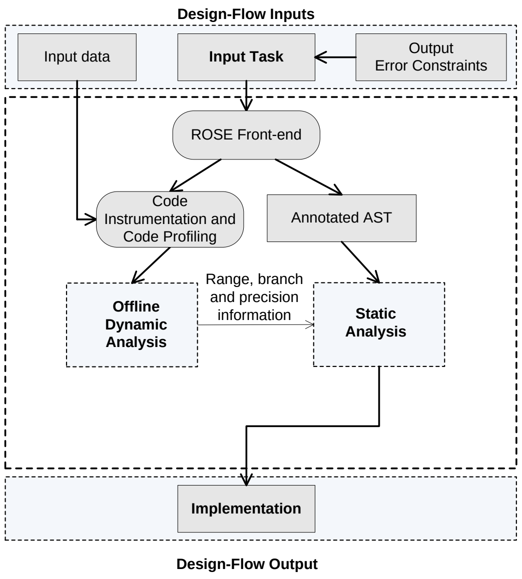


Fig. 6: An overview of the data representation optimisation design flow.

```
#pragma data_opt range(x_in, 5, 10)
#pragma data_opt range(y_in, 5, 10)
#pragma data_opt precision(z_out,6)

double x_in, y_in
double z_out;

z_out = x_in * y_in;
```

Listing 4: The data representation optimisation process requires as inputs: (1) the ranges of all input variables, and (2) the desired precision for the output variables. This information can be derived automatically from profiling information.

```
#pragma data_opt precision(x_in,  11)
#pragma data_opt precision(y_in,  11)
#pragma data_opt precision(z_out, 11)

z_out = x_in * y_in;
```

Listing 5: The data representation optimisation process outputs the precision of all variables in the program. This information can be used to synthesise resource efficient FPGA designs.

E. Data Representation Optimisation

This section describes the data representation optimisation process used in Harmonic. Currently this is an FPGA-specific optimisation built in ROSE, and part of our task transformation engine.

The goal of our data representation optimisation is to allow users to trade accuracy of computation with performance metrics such as execution time, resource usage and power consumption [16]. In the context of reconfigurable hardware, such as FPGAs, this means exploiting the ability to adjust the size of each data unit on a bit-by-bit basis, as opposed to instruction processors where data must be adjusted to be compatible with register and memory sizes (such as 32 bits or 64 bits).

Fig. 6 presents a simplified design-flow for generating resource efficient designs. It can be split into two parts: static analysis and dynamic analysis.

Static analysis takes as input (Listing 4): (1) the ranges of all input variables, and (2) the error constraints for the output variables (along with any constraints imposed on intermediate variables). The system analyses the design representation in the form of an abstract syntax tree (AST), and stores all relevant information in cost and error tables. Once the AST has been analysed, we proceed to range analysis which makes use of interval arithmetic in computing the maximum and minimum values of variables and expressions in the program. It is important to note that the range of a variable is stored with the specific instance of the variable because it can be different at each point in the program. At the end of the range analysis phase, we perform precision analysis to compute the accuracy required for each variable and expression of the program.

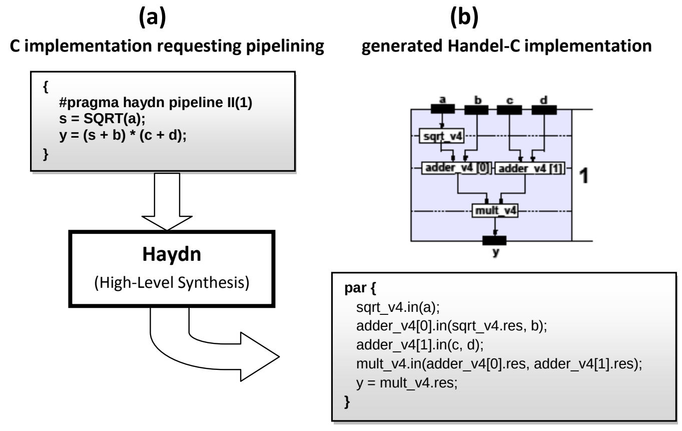
If input information cannot be supplied, a dynamic analysis can be run to determine the input ranges, and output precision requirements automatically. In addition, the code can be instrumented to uncover other dynamic information such as the number of iterations of a loop. This way, the dynamic mode is able to generate less conservative results than the static mode, since it does not have to assume that variables inside a loop have the maximum range, and therefore there is more scope to reduce the hardware design area.

At the end of this process, an implementation is generated where the code is annotated with range and precision of required variables (Listing 5). This information can then be supplied to the FPGA synthesis tool to generate efficient designs.

In addition to generating resource efficient designs, we extend our approach to reduce power consumption of circuits using an accuracy-guaranteed word-length optimisation. We adapt circuit word-length at run time to decrease power consumption, with optimisations based on branch statistics. Our tool uses a technique related to Automatic Differentiation to analyse library cores specified as black box functions, which do not include implementation information. We use this technique to analyse benchmarks containing library functions, such as square root. Our approach shows that power savings of up to 32% can be achieved by reducing the accuracy from 32 bits to 20 bits. Some benchmarks we adopt cannot be processed by previous approaches, because they do not support black box functions.

F. Hardware Synthesis

Tasks that are suitable for hardware implementation are mapped to FPGAs using the Haydn compiler [11], which performs high-level hardware synthesis (Fig. 1).



```
{
    #pragma haydn pipeline II(1)
    s = SQRT(a);
    y = (s + b) * (c + d);
}
```

```
par {
    sqrt_v4.in(a);
    adder_v4[0].in(sqrt_v4.res, b);
    adder_v4[1].in(c, d);
    mult_v4.in(adder_v4[0].res, adder_v4[1].res);
    y = mult_v4.res;
}
```

Fig. 7: An illustration of the hardware synthesis process. The task transformation engine may add source-annotations to certain kernels (such as loops) to indicate that certain transformations are required for FPGA. In **(a)** we show a source-annotation that requires a fully pipelined design (initiation interval of 1). The result of the high-level synthesis process is shown in **(b)**. The Handel-C code contains a specific construct to denote explicit parallelism. Users can modify this code to fine-tune the design.

The strength of our approach is that we combine behavioral and cycle-accurate (RTL) methodologies in order to achieve the best tradeoff between design-time and design quality.

At a first stage, the task transformation engine derives one or more implementations from a task description. These implementations are the result of transformations that may help hardware pipelining, such as loop interchange and loop unrolling, as well as `#pragma` annotations that are placed to trigger hardware pipelining, as shown in Fig. 7(a). Along with these annotations, the task transformation engine can also set the optimisation options, such as the degree of parallelism and resource sharing.

Once the mapper selects the appropriate C implementation, as shown in Fig. 7(a), to run on the FPGA, we use the Haydn tools to synthesise it and generate the corresponding Handel-C code, as shown in Fig. 7(b). Unlike the initial C implementation which captures the behaviour of the design, the Handel-C implementation captures its structure, which can be modified to explore, for instance, trade-offs between performance and resource utilisation. Finally, the Handel-C implementation can be compiled to netlist and then to bitstream using the place and route tools provided by FPGA vendors.

G. Experimental Features

This section provides an overview of some of the new experiments we have been applying to Harmonic.

Cost estimation. Section III-B mentions that each processing element targeted by our toolchain needs to provide a processor driver which can be used in estimating the costs of executing an arbitrary task on the processing element. The accuracy of the estimator is important, since it affects the mapping selection. Our approach for estimating the cost of a task on a particular processing element currently exploits rule-based techniques. Our rule-based estimator makes use of linear regression to estimate the cost based on a set of metrics:

$$EstTime = \sum_{i=1}^{N} T_{P_i} \quad (1)$$

where N is the number of instructions, P_i is the type of instruction i, T_{P_i} is the execution time of instruction P_i. Each processing element contains one set of T_{P_i} for each type of instruction. Instructions include conditionals and loops, as well as function calls. Other approaches for cost estimation, such as those based on neural networks, are also being explored.

Automatic verification. A verification framework has been developed in conjunction with the task transformation engine [17]. This framework can automatically verify the correctness of the transformed code with respect to the original source, and currently works for a subset of ANSI C. The proposed approach preserves the correct functional behaviour of the application using equivalence checking methods in conjunction with symbolic simulation techniques. The design verification step ensures that the optimisation process does not change the functional behaviour of the original design.

Model-based transformations. The task transformation engine (Section III-C) supports pattern-based transformations, based on recognising and transforming simple syntax or dataflow patterns. We experiment with combining such pattern-based transformations with model-based transformations, which map the source code into an underlying mathematical model and solution method. We show how the two approaches can benefit each other, with the pattern-based approach allowing the model-based approach to be both simplified and more widely applied [18]. Using a model-based approach for data reuse and loop-level parallelisation, the combined approach improves system performance by up to 57 times.

IV. Examples

This section includes two examples to illustrate how facilities for task transformation and mapping selection in Harmonic can be used in optimising heterogeneous designs.

A. Task Transformation

To show the effect of our transformation engine, we apply a set of transformations to an application that models a vibrating guitar string. These transformations have been described in both CML and ROSE, and allow the user to explore the available design space, optimising for speed and memory usage. We modify the application for a 200 second simulated time to show the difference between the various sets of transformations. The set of transformations includes:

- **S:** simplify (inline functions, make iteration variables integer, recover expressions from three-address code)
- **I:** make iteration bounds integer
- **N:** normalise loop bounds (make loop run from 0 to N-2 instead of 1 to N-1)
- **M:** merge two of the loops
- **C:** cache one array element in a temporary variable to save it being reread
- **H:** hoist a constant assignment outside the loop
- **R:** remove an array, reducing 33% of memory usage (using two arrays instead of three)

Fig. 8 shows how the design space can be explored by composing these transformations. Transformation S provides an almost three-fold improvement, mostly by allowing the compiler to schedule the resulting code. Transformation I gives nearly another two-fold improvement, by removing floating-point operations from the inner loop. Transformation N gives a small improvement after transformation I. Transformation M slows the code down, because the merged loop uses the GPP cache badly. Transformation C improves the integer code (I) but leaves the floating point version unimproved. Finally, transformation R gives a small improvement to the integer version, but actually slows down the floating-point version. Overall, we have explored the design space of transformations to improve execution time from 61.5 seconds to 9.8 seconds, resulting in 6.3 times speedup.

As described in Section III-G, we have developed a tool that checks the functional correctness of the transformed design with respect to the original design; examples of using this tool can be found in [17].

B. Mapping Selection

The reference heterogeneous computing system in this example contains three processing elements: a GPP, an FPGA, and a DSP; these three components are fully connected. Specifically, the GPP is an Intel Pentium-4 3.2GHz microprocessor, the FPGA is a Xilinx Virtex-II XC2V6000 device, and the DSP is an Atmel mAgic floating-point DSP.

Each processing element has a local memory for data storage during task execution. Results of a task's predecessors must be transferred to the local memory before this task starts execution. The method proposed is not limited to this architecture; this reference architecture is used on this occasion to illustrate the performance of difference approaches. Three applications are employed: FIR filtering, matrix multiplication, and hidden Markov model (HMM) decoding for pattern recognition. The number of tasks involved are 102, 112, and 80 respectively.

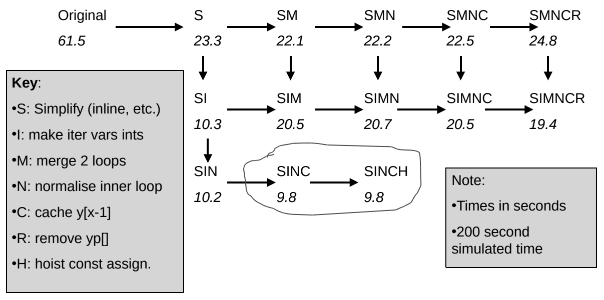


Fig. 8: Starting with the original code for the application that models the vibration of a guitar string, we explore ways of using seven different transformations to attempt to improve the run time and memory usage. Much of the speedup comes from simplifying the code and making iteration variables integer, while the remainder comes from caching to prevent a repeat memory access and removing a constant assignment from the loop body. The caching also enables one array to be eliminated (about 33% reduction in memory usage), possibly at the expense of performance.

Figure 9 shows the speedup comparison between the proposed mapping/scheduling approach and two other approaches: a system that performs mapping and scheduling separately [20], and an integrated system which uses a single neighborhood function [21].

Our approach can achieve more than 10 times speedup than using a single microprocessor: a speedup of 11.72 times is obtained for HMM decoding. Furthermore, it outperforms the other two approaches in all cases – the improvements over the separate approach [20] are 18.3%, 13.2% and 4.5% respectively, and the corresponding improvements over the integrated approach [21] are 199%, 316%, and 235%. The improvement is more pronounced than [21] which has a single neighborhood function with a strategy of choosing the best neighbor at each iteration. This shows that using and designing multiple neighborhood functions carefully are crucial.

The separate approach [20] is mapping dominated; it searches for the best mapping for a particular scheduling method. Since tasks are relatively similar in each application, they are likely to be mapped to the same processing element. One can observe that the improvement of the proposed integrated approach over the separate approach [20] for the HMM application is less significant than the FIR and the matrix applications; one reason is that the amount of data flow is smaller in the HMM application, so the penalty of inappropriate task mapping using the separate approach is also less significant.

Further examples of our mapping selection tool can be found in [22].

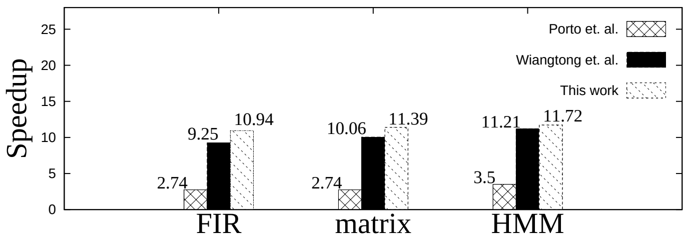


Fig. 9: Speedup comparison for various applications. *Wiangtong*: an approach that separate mapping and scheduling [20]. *Porto*: an integrated approach using single neighborhood function [21].

V. SUMMARY

An effective toolchain is key to productivity of application developers targeting heterogeneous systems. The Harmonic toolchain has several novel features for improving design quality while reducing development time. The core tools in Harmonic include a task transformation engine, a mapping selector, a data representation optimiser, and a hardware synthesiser. The experimental tools in Harmonic include facilities for cost estimation, design verification, and combination of model-based and pattern-based transformations.

The modular structure of the Harmonic toolchain simplifies its customisation. Such customisation enables Harmonic to be tailored for different application domains; it also allows Harmonic to evolve with technological advances and with changes in application requirements. Indeed Harmonic has proved successful in providing an infrastructure that facilitates rapid experiments of ideas for new techniques in enhancing quality and productivity of heterogeneous system design.

We hope that Harmonic will contribute to both foundation and applications of next-generation heterogeneous systems. Current and future work includes refining the Harmonic toolchain to provide a basis both for a research tool to support design automation experiments and benchmarking, and for a platform on which industrial-quality tools can be developed. Various extensions of Harmonic are also being explored, to support targets ranging from high-performance computers to embedded system-on-chip devices.

ACKNOWLEDGMENTS

The support of FP6 hArtes (Holistic Approach to Reconfigurable Real Time Embedded Systems) Project, the UK Engineering and Physical Sciences Research Council, HiPEAC and Xilinx is gratefully acknowledged. We thank George Constantinides, Matti Juvonen and Philip Leong for their contributution to the material described in this paper.

REFERENCES

[1] A. Hamann et al, "A Framework for Modular Analysis and Exploration of heterogeneous embedded Systems", *Real-Time Systems*, vol. 33, no. 1–3, pp. 101–137, July 2006.

[2] T. El-Ghazawi et al, "The Promise of High-Performance Reconfigurable Computing", *Computer*, pp. 69–76, February 2008.

[3] "3L Diamond User Guides," http://www.3l.com/user-guides.

[4] "Gedae Manuals," http://www.gedae.com/doc_manuals.php.

[5] "CleanC Code Optimization," http://www2.imec.be/atomium.

[6] T. Stefanov, C. Zissulescu, A. Turjan, B. Kienhuis, and E. Deprette, "System Design using Khan Process Networks: the Compaan/Laura approach," in *Proceedings of Design, Automation and Test in Europe (DATE)*, 2004, pp. 340–345.

[7] "Hy-C compiler," http://www.cse.unt.edu/~sweany/research/hy-c/.

[8] B.P. Dave, G. Lakshminarayana, and N.K. Jha, "Cosyn: Hardware-software Co-synthesis of Heterogeneous Distributed Embedded systems," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 7, no. 1, pp. 92–104, 1999.

[9] M. Schordan and Dan Quinlan, "A Source-To-Source Architecture for User-Defined Optimizations", in *Modular Programming Languages*, LNCS 2789, Springer, 2003, pp. 214–223.

[10] M.J. Quinn, *Parallel Programming in C with MPI and OpenMP*, McGraw-Hill, 2004.

[11] J.G.F. Coutinho, J. Jiang and W. Luk, "Interleaving Behavioral and Cycle-Accurate Descriptions for Reconfigurable Hardware Compilation", in *Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines*, IEEE Computer Society Press, 2005, pp. 245–254.

[12] Atmel, *Diopsis 940HF, ARM926EJ-S Plus One GFLOPS DSP*, July 2008.

[13] I. Colacicco, G. Marchiori and R. Tripiccione, "The Hardware Application Platform of the hArtes Project", in *Proceedings of International Conference on Field Programmable Logic and Applications*, 2008, pp. 439–442.

[14] F. Ferrandi et al, "Partitioning and Mapping for the hArtes European Project", in *Proceedings of Workshop on Directions in FPGAs and Reconfigurable Systems: Design, Programming and Technologies for Adaptive Heterogeneous Systems-on-Chip and their European Dimensions*, *Design Automation and Test in Europe Conference*, 2007.

[15] Y.D. Yankova et al, "DWARV: Delft Workbench Automated Reconfigurable VHDL Generator", in *Proceedings of International Conference on Field Programmable Logic and Applications*, 2007, pp. 697–701.

[16] W.G. Osborne, J.G.F. Coutinho, W. Luk and O. Mencer, "Power and Branch Aware Word-Length Optimization", in *Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines*, IEEE Computer Society Press, 2008, pp. 129–138.

[17] K. Susanto, W. Luk, J.G.F. Coutinho and T.J. Todman, "Design Validation by Symbolic Simulation and Equivalence Checking: A Case Study in Memory Optimisation for Image Manipulation", in *Proceedings of 35th Conference on Current Trends in Theory and Practice of Computer Science*, 2009.

[18] Q. Liu et al, "Optimising Designs by Combining Model-based transformations and Pattern-based Transformations", in *Proceedings of International Conference on Field Programmable Logic and Applications*, 2009.

[19] Y.M. Lam, J.G.F. Coutinho, W. Luk, and P.H.W. Leong, "Integrated Hardware/Software Codesign for Heterogeneous Computing Systems," in *Proceedings of the Southern Programmable Logic Conference*, 2008, pp. 217–220.

[20] T. Wiangtong, P.Y.K. Cheung, and W. Luk, "Hardware/Software Codesign: A Systematic Approach Targeting Data-intesive Applications," *IEEE Signal Processing Magazine*, vol. 22, no. 3, pp. 14–22, May 2005.

[21] S.C.S. Porto and C.C. Ribeiro, "A Tabu Search Approach to Task Scheduling on Heterogeneous Processors under Precedence Constraints," *International Journal of High-Speed Computing*, vol. 7, pp. 45–71, 1995.

[22] Y.M. Lam, J.G.F. Coutinho, W. Luk, and P.H.W. Leong, "Mapping and Scheduling with Task Clustering for Heterogeneous Computing Systems", in *Proceedings of International Conference on Field Programmable Logic and Applications*, 2008, pp. 275–280.