POLITECNICO DI TORINO
Repository ISTITUZIONALE

Energy-aware parallelization flow and toolset for C code

Original
Energy-aware parallelization flow and toolset for C code / Lazarescu, M.T., Albert, C., Adrien, G., Nhat Minn, L.ê., Lavagno, L., Antoniu, P., Manuel, P., Andrei, T., Alexandru, S.. - ELETTRONICO. - (2014), pp. 79-88. (17th International Workshop on Software and Compilers for Embedded Systems - SCOPES '14 New York (USA) 2014) [10.1145/2609248.2609264].

Availability:
This version is available at: 11583/2565955 since: 2020-10-22T14:39:14Z

Publisher:
ACM

Published
DOI:10.1145/2609248.2609264

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright
ACM postprint/Author's Accepted Manuscript, con Copyr. autore

(Article begins on next page)

16 October 2026

Energy-aware parallelization toolset and flow for C code

The PHARAON FP7 project

Mihai T. Lazarescu
Politecnico di Torino,
Turin, Italy
mihai.lazarescu@polito.it

Albert Cohen
INRIA and École Normale
Suprieure, Paris, France
albert.cohen@inria.fr

Luciano Lavagno
Politecnico di Torino,
Turin, Italy
luciano.lavagno@polito.it

Antoniu Pop
INRIA and École Normale
Suprieure, Paris, France
antoniu.pop@inria.fr

Manuel Prieto
Tedesys Global S.L.,
Santander, Spain
mprieto@tedesys.com

Andrei Terechko
Vector Fabrics, Eindhoven,
The Netherlands
andrei@vectorfabrics.com

ABSTRACT

Multicore architectures are increasingly used in embedded systems to achieve higher throughput with lower energy consumption. This trend accentuated the need to convert existing sequential code to effectively exploit the resources of these architectures.

We present the work-in-progress of the EU FP7 PHARAON project that aims to develop a complete set of techniques and tools to guide and assist the designer in the development process for heterogeneous parallel architectures. We focus on the legacy C code parallelization flow that includes a performance estimation tool, a parallelization tool, and a streaming-oriented parallelization framework. We demonstrate the effectiveness of the use of the toolset on a use case where we measure the quality and time for parallelization for inexperienced users and the parallelization flow and performance results for the parallelization of a practical example of a stereo vision application.

Categories and Subject Descriptors

D.1 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*

General Terms

execution profiling, data dependency analysis, program parallelization, energy estimation

1. INTRODUCTION

Market evolution over the last years shows a significant increase of the use of multicore architectures in new projects [4]. Over the last decade, processors and systems refocus from the acceleration of the execution of a single-thread to the increase of the overall throughput by means of multiprocessor architectures. Also, multicore architectures traditionally used in specific domains with very high processing needs gradually permeated to many embedded systems increasing the need to parallelize massive amounts of legacy sequential code [3, 9]. However, even when parallelism is taken into account from the start of a project, writing programs for efficient execution on parallel architectures is still considered a challenging task [10, 2].

The revolution in hardware architectures challenges the software development techniques to efficiently exploit the potential of the multicore architectures, including the performance-power trade-offs that are often important for portable embedded systems. Automated software parallelization has been extensively explored especially at the statement, basic block and loop levels, which are appropriate for VLIW and vector processors [20, 8]. By contrast, the tools for exploring the parallelization opportunities at the *task* level, which are best suited for modern multi-core processors, were less explored, with some notable exceptions [14, 6]. However, most of the latter techniques are so far restricted to specific types of loops and data access patterns.

The PHARAON project aims to enable the development of complex systems with high processing needs and low-power requirements. Figure 1 shows the techniques and tools developed to this end.

The first set addresses the design flow, starting from UML/-MARTE[1] specifications up to implementation on multicore platform. It assists the design space exploration for for the best software architecture and for parallelization opportunities. The second set addresses the run-time adaptation of platform performance (e.g., frequency and voltage) to minimize the energy consumption.

1.1 Evolution beyond the state of the art

Although long studied, compilers can generally extract a limited level of parallelism unless they are used for special applications and, often, for specific coding styles [14, 13]. Efforts like MORPHEUS [24], CRISP [1] and MEGHA [22]

[1]UML Profile for MARTE `http://www.omgmarte.org/`

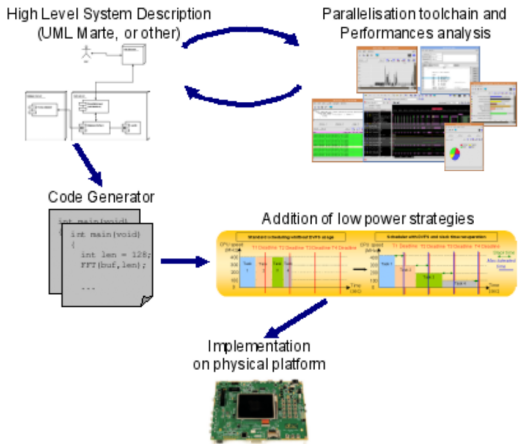

Figure 1: PHARAON global approach and tools interactions

are usually tailored to the target architecture they produce parallel code for. Dominant industry players have also proposed several compilation and debugging tools. For example, OpenCL and CUDA extend the C language to generate efficient code for GPUs. In this project, OpenStream extended the OpenMP standard, better suited for CPUs, with streaming-oriented constructs.

UML is a common modeling language for high-level system design [19]. Previous works propose semi-automatic generation of HW/SW infrastructures [5] and a flow to dynamically reconfigurable SoCs [12]. Low-power run-time management and scheduling have also been proposed, most recent using dynamic voltage and frequency scaling [7, 11]. Design-time approaches use slow integer linear programming [23] and cannot be used at run time. PHARAON proposes a complete framework addressing heterogeneous multi-processor platforms for power consumption optimization.

1.2 System design flow

Figure 2 shows the flow that extends from the high-level UML specifications to the programming of the target platform. UML specification allows to handle homogeneous, heterogeneous and distributed systems and explore the parallelization between components though automatic code generation. This code is used to perform performance and parallelization analysis, code synthesis and power management to optimize the use of target platform resources.

First stage uses the Pareon performance simulator to evaluate the timing and energy of the C code of the UML components. The second stage is driven by the interactive parallelization tool ParTools that is used to discover the parallelization opportunities. The optimized code is either simulated again or implemented and analyzed onto the physical platform in the third stage, in order to assess both the parallelization quality and to extract information for the run-time optimizations. These are used by the reconfiguration manager and low-power scheduler are deployed on the physical platform to provide the required application performance with reduced power consumption.

2. PHARAON WORKFLOW FOR PARALLELIZATION

Parallelizing an existing sequential implementation guided only by a classical source code profiler is not trivial *without prior knowledge of the software*. For instance, a typical `gprof` profile shown in Listing 1

Listing 1: Typical execution profile (`gprof`) output

```
  %   cum. self     self   total
 time   sec. sec.     calls s/call  s/call name
16.61   2.79 2.79 788215425   0.00    0.00 Dot
13.78   5.12 2.32 141631877   0.00    0.00 IntersectQuad
 8.26   6.50 1.39 281277610   0.00    0.00 intersectObject
 8.02   7.86 1.35 139645733   0.00    0.00 IntersectSphere
 7.90   9.19 1.33  69361053   0.00    0.00 NormalizeVec3
...
```

shows clearly the most computation-intensive parts of the program, but does not provide any information on data flows and dependencies, which are well known as one of the most important parallelization inhibitors. More advanced tools provide more details, but still do not cover the data dependencies within the whole program.

For these reasons, the PHARAON workflow for parallelization collects program-wide data dependencies at run-time and presents them for analysis in an abstracted and intuitive way. The toolset flow does not make any specific assumptions on the developer skills, parallelization method, syntax or parallelization framework:

- it starts with run-time collection of execution profile and data dependencies of the serial program;
- performance analysis either simulated or from the (embedded) target system to collect energy consumption estimates and execution histograms;
- display in an intuitive and interactive graphical form of the execution profile, data dependencies, and performance estimations;
- manual analysis of the data and selection of the most promising parallelization opportunities and style;
- parallelization, test and debug of the parallel code, and measurement of performance enhancements;
- code refactoring to improve the parallelization performance of the algorithms.

The steps above can be iterated as needed until are achieved satisfactory results with the effort allocated to the project. In the following sections, the PHARAON toolset components that support the flow will be presented in more detail. Then, the effectiveness of the use of the toolset will be demonstrated, both in terms of simplification of the parallelization task for low skill users as well as the acceleration obtained on a stereo vision application of practical interest.

2.1 ParTools parallelization toolset

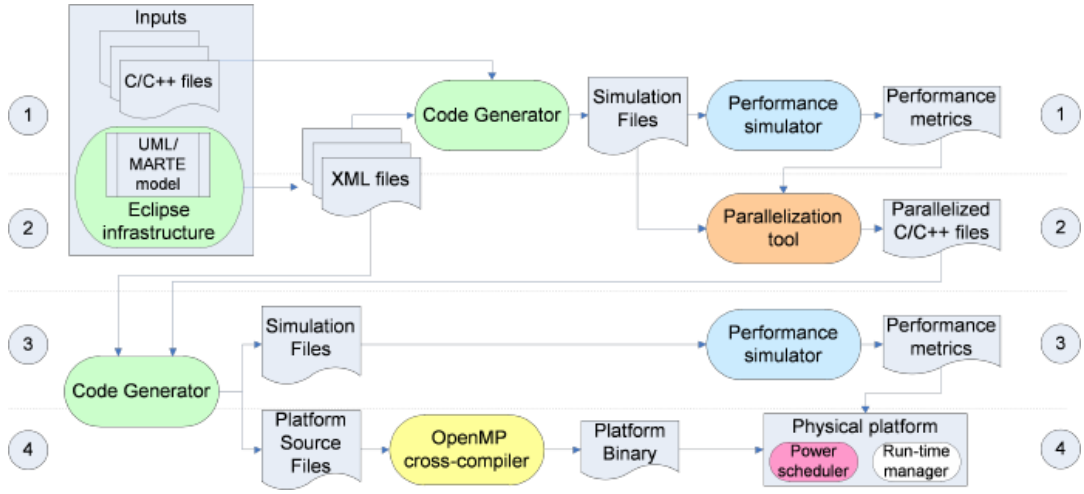

Figure 2: PHARAON design flow

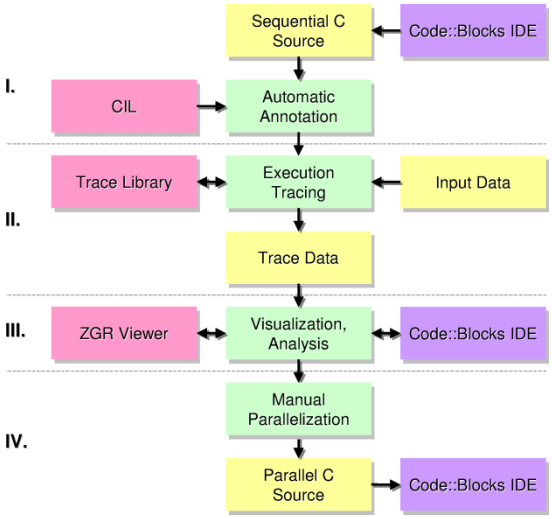

Figure 3: ParTools toolset parallelization flow

ParTools[2] [16, 15] is a free software project designed to support the developers of various skill levels to parallelize legacy sequential C code that can include complex control structures, pointer operations, and dynamic memory allocation. ParTools was designed to facilitate the discovery of both task and data parallelization opportunities and can be used for any parallelization technique.

The toolset flow, shown in Figure 3, is divided in four stages: *I* source instrumentation, *II* run-time execution trace profile and data dependency collection and compaction, *III* graphical visualization and analysis of execution data, and *IV* source code parallelization. Its operation is controlled from the Code::Blocks IDE.

An automatic annotator instruments in stage *I* the sequential source for run-time data dependency collection. The data generated by the instrumentation are collected and compacted at run-time in stage *II* by a library, and saved in the project at the end of the execution. It is graphically displayed in stage *III* as a data dependency graph (DDG), with the nodes representing program control (e.g., statements, loops, function calls) and the edges representing the data dependencies. All elements are analyzed based on their execution call stacks, since their execution parameters can change with the context. The nodes for complex program structures (e.g., loops, function calls) fold all the execution call stacks rooted there. These can be unfolded progressively, as needed to discover good parallelization candidates, as we will show later. Stage *IV* supports manual program parallelization based on above exploration. The source code in the IDE is connected with the graph elements in the graph viewer. Also, the graph viewer provides several methods to temporarily hide graph sections that are not relevant for the parallelization, such as graph re-rooting to any given node.

ParTools analysis can complement automatic parallelization tools (e.g., that of Compaan Design[3]) which can significantly benefit from the toolset-driven *program-wide data dependency analysis*. ParTools can show: where the compute intensive procedures are; if there are any data dependencies besides those through procedure arguments; whether the procedure inputs and outputs are truly unaliased; whether the procedure inputs are truly read-only and outputs are truly write-only. Also, ParTools can import data from external analysis tools that complement its analysis capabilities. For example, energy analysis and execution histograms from Pareon can be imported and displayed on the graph to provide the developers with a more comprehensive view on program execution to make better parallelization decisions.

Graphical visualization opens by abstracting all execution details under the call to `main()` function, as shown in Figure 4. The fold label shows the fold type, its estimated execution load and energy consumption (imported from the Pareon analyzer), the source file name and line, the function name followed by its unique call stack ID. The folds can be unfolded one level at a time to help the developer uncover

[2]ParTools project: `http://sf.net/projects/partools/`

[3]Compaan Design BV `http://www.compaandesign.com/`

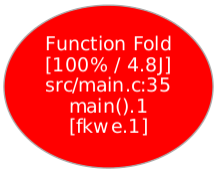

Figure 4: ParTools initial view folds all execution and dependencies under the main() function.

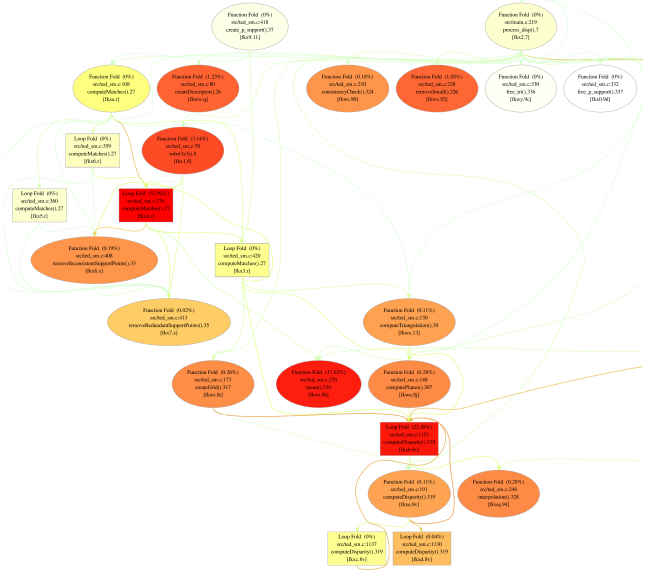

Figure 5: Analysis of a stereo vision application. The two loop folds (square shape) with stronger colourization include 53% and 18% of program execution and may be good parallelization candidates with no strong data dependencies between them.

data relevant for discovery of parallelization opportunities. For instance, Figure 5 shows an expanded view in the analysis of the stereo vision application shown in Figure 4, in which the rectangular nodes correspond to loop folds and the node labels show data similar to that shown in Figure 4.

The data dependency view of a selected DDG node is another important feature of the toolset to help the parallelization decisions. It can be generated for any fold node and shows in detail the read and write data dependencies of that fold, which are essential for any parallelization mechanism, language and style. The view is organized in layers, as shown in the excerpt in Figure 6. The top layer displays the leaf nodes (C statements) that produce the incoming data, which are displayed on the next layer. The middle layer displays the statements in the selected fold that consume data from or produce data for outside the fold, which are shown on the next layer. The bottom layer displays the leaf nodes from outside the fold that consume the data produced within it. These dependencies are typically difficult to extract through static code analysis or inspection, since the producers and consumers can be at various depths and in different call stacks, and transfer data through any type of data (dynamic, local, global, etc.)

```
            for(y=0; y<blockSize; y++)
                for(x=0; x<blockSize; x++)
// The following statement has:
//
// * input data dependencies:
//   light2Obj, c, img, image, blueSphere, redSphere, blueSphere0bj,
//
// * output data dependencies:
//   img->m_buffer
//
// OpenMP pragma template:
// #pragma omp task shared(light2Obj, c, img, image, blueSphere, red
                    RenderPixel(scene, x, y, image, c);
```

Figure 7: Data dependencies and OpenMP pragma template inserted as comments in the source code.

To further help the developer, the toolset can insert comments in the source code with the data dependencies and an OpenMP pragma template that can be adapted for the parallelization of the fold node of interest (see Figure 7).

2.2 Performance analysis in Pareon

2.2.1 Performance analysis toolflow

The Pareon tool-suite features leading-edge analysis and interactive parallelization capabilities. In the PHARAON project, it provides the analysis of the performance of C and C++ applications on the target hardware platforms (currently ARM Cortex A9 and an Intel Core 5), including energy consumption estimation. These data are then imported by the parallelization tool (ParTools) to provide the developer with a comprehensive view of the run-time effects of the program, to help them making effective parallelization decisions. The energy estimates are also used by the low power scheduler to select the most power-efficient operating mode of the system. Pareon can also analyze parallel C and C++ programs that use POSIX threads or OpenMP pragmas (the latter are under test), which allows to check the parallelization decision effects and close the loop of the PHARAON toolset[4].

The internal Pareon flow for performance analysis is shown in Figure 8. Pareon offers both command line interface (CLI) tools and a GUI. The CLI tools are used to automate the interface with the PHARAON project toolset, while the GUI allows the developers to inspect the results of the modelling. The `vfcc` compiler is one of the most important CLI tools that translates the source code into a generic executable for a target-independent intermediate instruction set architecture. This code is then run by the Pareon simulator using the necessary test data, input files, environment variables, etc. to collect various statistics. These can then be converted into estimates for a particular hardware target platform using report commands.

Pareon performance analysis is only a few hundred times slower than native execution, which is much faster than the usual gate-level back-annotated timing and power modelling tools in the EDA industry. Due to architecture virtualization, Pareon can model configurations that do not exist (yet) as hardware components, e.g., using more processor cores, or easily perform design space exploration by changing the platform or the operating conditions for the simulation step.

[4]Extensive Pareon documentation is available online at `http://www.vectorfabrics.com/docs/pareon/current/`

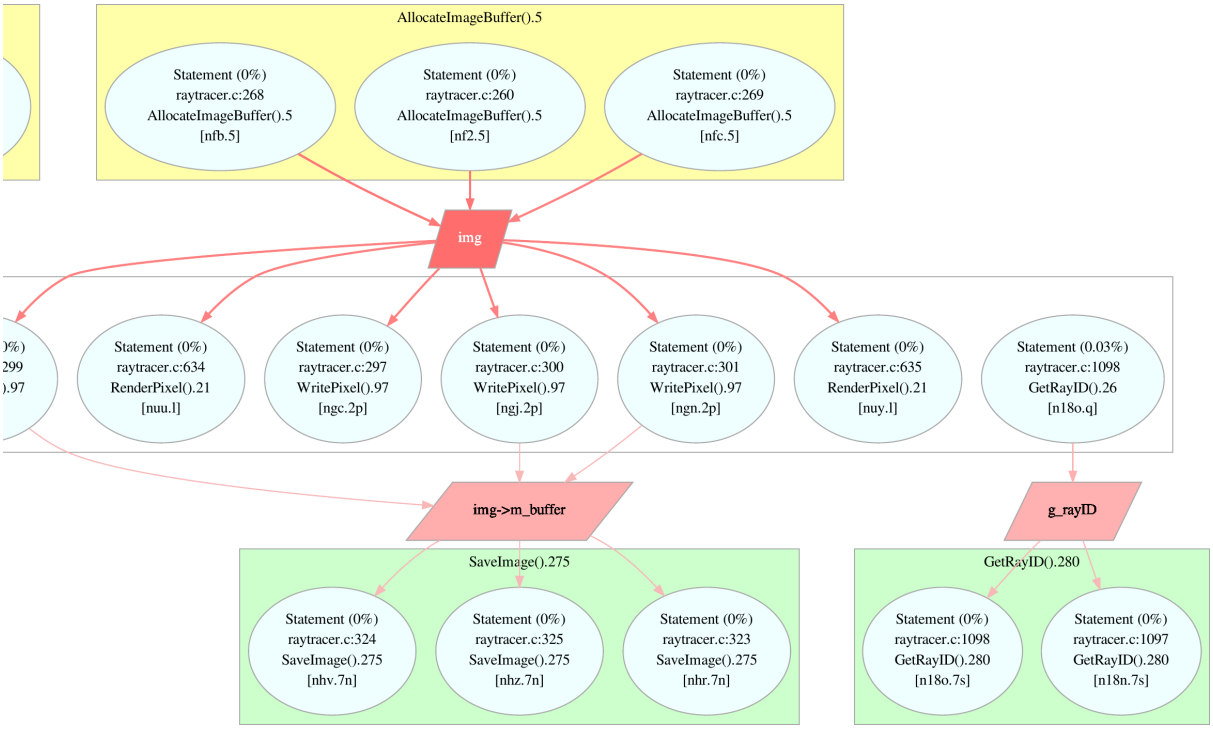

Figure 6: Excerpt of the data dependency view

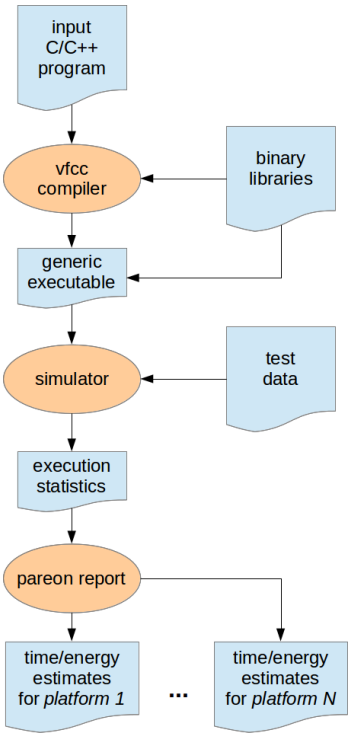

Figure 8: Pareon performance analysis toolflow

2.2.2 *Performance histograms*

Pareon performance analysis can generate timing and invocation count statistic histograms for functions and loops at run-time. These are important for parallelization decisions, since the execution parameters of the same code may depend on the context (e.g., on function arguments). For example, the loop body in Listing 2 has a variable execution time depending on the function argument. Thus, parallelizing the loop in the `main()` function using a round-robin scheduling is inefficient, since the invocation time is not constant. This would lead to imbalanced load and low speedups with respect to a dynamic scheduling.

Listing 2: Varying function timing

```
int foo(int n)
{
    int s = 0, i;
    for (i = 0; i < n; i++)
        s += i * i;
    return s;
}

int main()
{
    int val[] = { 5, 4, 2, 3, 4, 4, 3, 5 };
    int i;
    for (i = 0; i < 8; i++)
        val[i] = foo(val[i]);
    return 0;
}
```

Pareon histograms can be explored using the tool GUI or can be exported for integration in other tools, e.g., to complement the call stack-based analysis of ParTools. E.g., Figure 9 shows the timing histogram for a loop in terms of

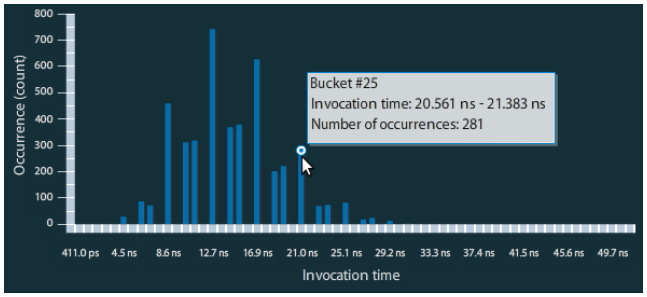

Figure 9: Pareon GUI with a timing histogram of a loop

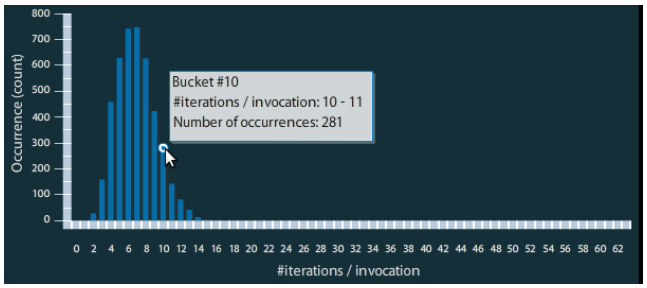

Figure 10: Pareon GUI with an iteration histogram of a loop

number of times it has been executed and how much time took each execution (grouped in time bins). The time bins can be explored further as shown in Figure 10 for bin #25 to look for specific patterns as follows. If there are multiple spikes with few iterations per invocation, the speedup is limited by the parallelization overhead of the loops with few iterations. One spike with many iterations per invocation generally benefits from parallelization. Loops with non-constant body execution time may benefit most from a dynamic scheduling to avoid workload imbalanced.

2.3 OpenStream: OpenMP extension for data-flow and stream parallelism

OpenStream[5] is a stream programming language, designed as an incremental extension to the OpenMP parallel programming language [21]. It allows expressing arbitrary task-level data flow dependence patterns through compiler annotations (pragmas) that dynamically generate a streaming program. The language supports nested task creation, modular composition, variable and unbounded sets of producers/consumers, and first-class streams. These features allow translating high-level parallel programming patterns into efficient data-flow code. OpenStream is provided as a tightly integrated collection of compilation, code generation, and concurrent runtime algorithms for task-level parallel programming, particularly effective on embedded multicores.

Data-flow execution is essential to reduce energy consumption, one of the primary focuses of the PHARAON project, by reducing the severity of the memory wall in two complementary ways: (1) thread-level data flow naturally hides

[5]OpenStream project `http://www.di.ens.fr/OpenStream`

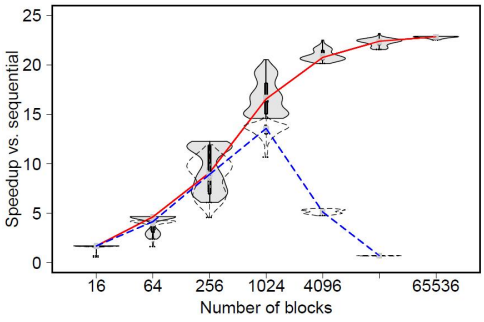

Figure 11: Speed-up comparison between OpenStream (solid) and StarSs (dashed) for block-sparse matrix LU factorization.

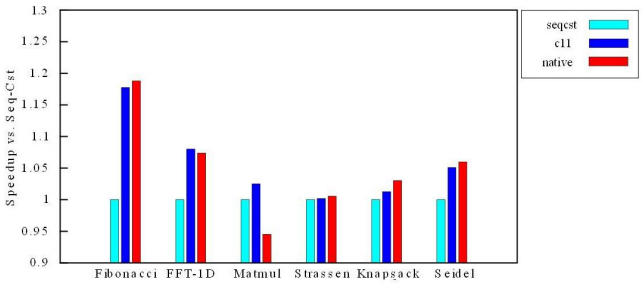

Figure 12: Speed-up comparison OpenStream and StarSs

latency and (2) decoupled producer-consumer pipelines favor on-chip communication, bypassing global memory. Furthermore, OpenStream exceeds the performance of state-of-the-art parallel programming environments like StarSs. Figure 11 shows comparatively that OpenStream speedups against sequential execution (solid) exceed those of StarSs (dashed) for a block-sparse matrix LU factorization on a dual-socket AMD Opteron Magny-Cours 6164HE machine with 2×12 cores at 1.7 GHz due to its optimized runtime for low-overhead synchronization and work-stealing scheduling that improves on Chase and Lev's concurrent doubly-ended queue. It has been ported to x86 and ARM architectures, the latter being optimized for its weak memory model leveraging on recent progress in memory consistency formalization as a first proof of the relaxed double-ended queue [17]. Figure 12) shows that the optimized ARM code generally outperforms the original sequentially consistent Chase–Lev in a variety of benchmarks, including a selection of standard fine-grained task-parallel computations.

OpenStream efficiently addresses another critical concurrent data structure for parallel languages and embedded multiprocessors, the single-producer, single-consumer (SPSC) FIFO queues. These may arise from a variety of parallel design patterns and from the distribution of Kahn process networks over multiprocessor architectures. With WeakRB [17] we focus on portability and correctness through concurrent implementation in C11 and performance through advanced caching and batching extensions, relaxed hypotheses on memory ordering, and leveraging the low-level atomics in C11 with relaxed memory consistency. We validate its portability and performance on 3 architectures with diverse

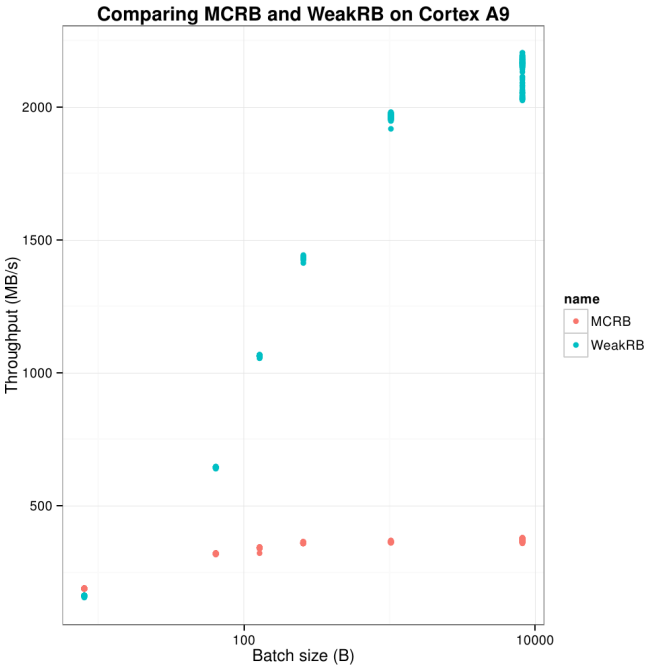

Figure 13: Comparison between MCRB and WeakRB on Cortex A9

hardware memory models, including 2 embedded platforms. Figure 13 shows how WeakRB outperforms one of the state of the art algorithms, MCRB [18], sustaining close-to peak throughput in core-to-core streaming communications.

2.4 Tool support: interface and automation

ParTools toolset is made of several free software projects integrated under the control of the IDE [16].

The parallelization flow described in Section 2 starts with the compilation of the sequential code using the `partools-cc` (a gcc-compatible wrapper for the C-to-C compiler that performs the code instrumentations needed for run-time profiling and tracing) and linked with the run-time tracing library using the `partools-ld` wrapper (a GNU linker wrapper). It is then run using a data set relevant for the application. These steps require minimal changes to make-based projects and can be fully automated. The profile data collected during the execution of the instrumented program is saved in the project and is read by the graph viewer when started by the IDE.

ParTools modular structure allows to easily integrate data from external tools, for example from the Pareon analysis tool as shown in the PHARAON parallelization flow in Section 2. The energy estimations from the Pareon toolset are displayed on the DDG to allow the developer to start the exploration for best parallelization opportunities in terms of execution speed-up and energy consumption reduction by focusing on: (1) folds that account for important parts of program execution, starting by unfolding the `main()` fold (see Figure 4) and (2) on important data flows that are highlighted in the DDG by prominent arrows. The high execution folds can be good candidates to data-parallelization, e.g., using OpenMP. At the same time, if these folds are connected by important data transfers, they may be good candidates for task-parallelization, e.g., using OpenStream.

Furthermore, the input and output data dependencies for the folds considered for parallelization can be analyzed using the detailed data dependency view shown in Figure 6. This view emphasizes the direction of the dependency (read or write), what source statements produce and consume it, where the variables holding them were declared, and if there are hidden data dependencies (e.g., on global data) that are not visible at the level in the call stack considered for parallelization.

3. EXPERIMENTAL EVALUATION

3.1 Comparative use test

To illustrate the benefits of the PHARAON toolset flow in this respect, we present the results of a comparative use test. Its purpose is to show how the use of the toolset helps relatively inexperienced users to more effectively parallelize a previously unknown legacy application. We used students from a second year course for the electronics engineering master (5th year overall) that covers modelling languages, such as SystemC, Esterel and Kahn Process networks, and the associated synthesis and verification algorithms and tools. The course does not teach specifically how to parallelize software, the students had only used the SystemC language to model multiple threads communicating via signals (i.e., using the Moore synchronous reactive model). They were also exposed to the concept of Kahn Process Networks, but had never written code using this computation model. Hence, the students entered the experiment without any knowledge of writing parallel software.

The test assignment was to analyze and parallelize three real-life use cases: an MJPEG encoder, a ray tracing algorithm, and a cascade of two FIR filters. The groups were partitioned in two sets, of which one was required to use only standard code analysis and development tools (such as `gprof` and OpenMP parallelization pragmas) and its results were used as baseline to assess the effects of using the toolset. The other set was required to use the PHARAON toolset in addition to the standard tools of the first set and its results were evaluated against the results of the first set separately, for each parallelization candidate program.

The students were requested to spend at most a couple of days on the parallelization. Only 9 groups out of 11 completed the assignment and the results of the test are summarized in Figure 14. The X axis lists the test cases as follows: "*mjpeg*" is an MJPEG encoding algorithm with an acyclic data dependency graph at the top level; "*FIR*" is a couple of cascaded FIR filters; "*raytracer*" is a ray tracing application with a well known top-level data parallelism. The tools used for the parallelization are: only standard development and analysis tools for "*no toolset*" and both standard and the PHARAON toolset for "*toolset*". The Y axis shows the time (in hours) needed to complete the various phases of the parallelization assignment, and the speedup obtained on a 4 core Intel architecture.

The graph indicates: the training time to get acquainted with the tools; the time to perform the first parallelization (discover parallelism, analyze the data dependencies, write

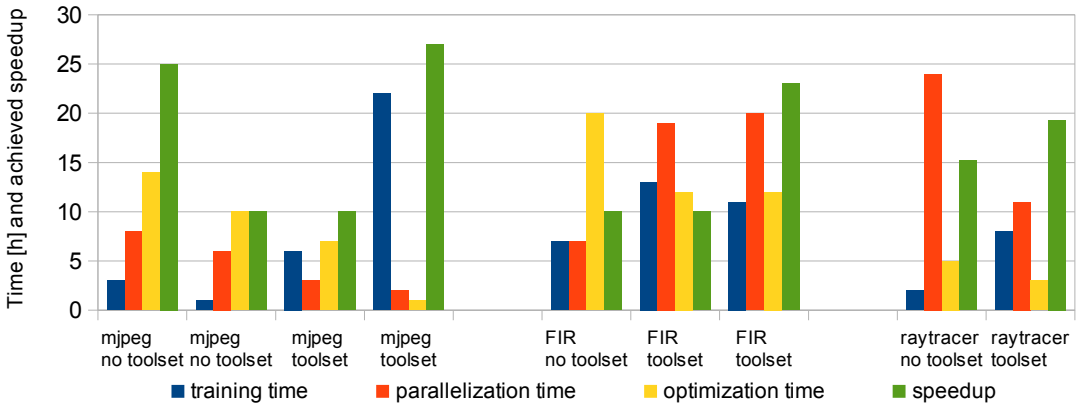

Figure 14: Results of the test of the toolset use for parallelization.

the parallel code using OpenMP pragmas, and debug the results so that the execution was correct); the time to further optimize the parallelized code to improve the speedup; and the final speedup to sequential execution.

The results of the "*mjpeg*" test show that using the toolset considerably reduced the parallelization time, but at the cost of more training time. Additionally, more time invested to learn the toolset appears to pay off by reducing the parallelization time later. The final speedup results are similar, with some variability that does not appear to depend on the use of the toolset. The "*FIR*" test shows that the group using the toolset was the only one obtaining any speedup. Learning how to use the toolset in this case took a long time. Also for the "*raytracer*" test, the use of the toolset reduced the parallelization time at the cost of more training time. The use of the toolset lead to slightly better speedup than without. However, neither group obtained a functionally correct parallelization since they missed some of the data dependencies due to the incompleteness of their code analysis. Up to a point this is unavoidable because of the "optimistic", trace-based, manual parallelization approach used. However, this prompted us to extended the toolset after the experiment with the capability to insert the data dependencies as comments in the source code as shown in Figure 7.

3.2 Stereo vision use case

Stereo vision applications infer 3D scene geometry from two images with different viewpoints by calculating a dense disparity or depth map from a pair of images under known camera configuration. The parallelization flow of the PHARAON toolset described in Section 2 was used on the application code as presented in Section 2.4.

Unfolding the highest level of abstraction shown in Figure 4 revealed that the fold of function `process()` call holds almost all program execution. Unfolding this one shows that function `process_disp()` folds 99.85% of program execution. Unfolding it reveals visually right away that three folds hold most of the execution load, as can be seen in the center of Figure 15: `computeMatches()` 53.11% (top), `computeDisparity()` 22.61% (bottom), and `mean()` 17.61% (right). We also notice that the data dependencies between these are not very strong suggesting that these folds may be suited for data-parallelism. In the corresponding source code, we find out that in fold `computeMatches()` the function `com-`

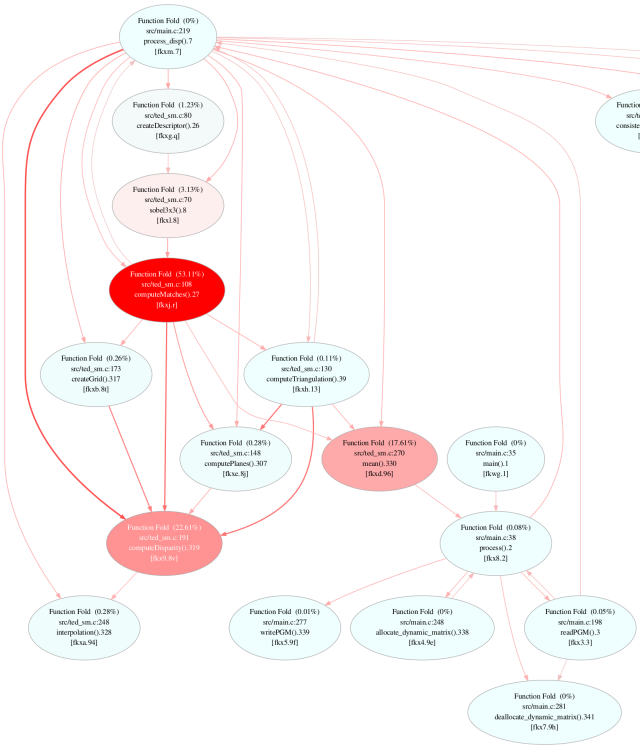

Figure 15: Unfold of folds `main()` 100% → `process()` 100% → `process_disp()` 99.85% for the stereo vision application shows three parallelization candidates: `computeMatches()` 53.11% (top), `computeDisparity()` 22.61% (bottom), and `mean()` 17.61% (right).

puteMatchingDisparity() is called twice within the body of the innermost of two nested loops, as shown in Listing 3 (without the leading pragma).

Listing 3: Contents of computeMatches() fold

```
#pragma omp parallel for
for (u_can=1; u_can<D_can_width; u_can++) {
  ...
  for (v_can=1; v_can<D_can_height; v_can++) {
    ...
    d=computeMatchingDisparity(&pu,&pv, ...
    if (d>=0) {
      ...
      computeMatchingDisparity(&pdif,&pv, ...
      ...
    }
  }
}
```

Analyzing the data dependency view, the loop histogram and the source code we deduce that the two calls to computeMatchingDisparity() are independent and that the iterations of the outer loop do not show major unbalances. Thus, its execution can be sliced and executed in parallel using the OpenMP pragma shown on top of Listing 3.

A similar analysis shows that the best parallelization for the other two candidates in Figure 15 (computeDisparity() and mean()) can be where they are called in process_disp(), as shown in Listing 4 for the former.

Listing 4: Call of computeDisparity() function

```
computeDisparity(p_support,tri_1, ...
computeDisparity(p_support,tri_2, ...
```

Dependency analysis shows that the two calls are independent and can be executed in parallel as shown in Listing 5:

Listing 5: Parallelization of computeDisparity() call

```
#pragma omp parallel sections
{
#pragma omp section
  {
    computeDisparity(p_support,tri_1, ...
  }
#pragma omp section
  {
    computeDisparity(p_support,tri_2, ...
  }
}
```

The call to mean() is analyzed and parallelized analogously.

The speedup of these parallelizations was measured on the target architecture composed of two Intel i5 cores running at 1.20GHz (i5-3230M) processing a set of images of 1024×768 pixels. The results are reported in Table 1 and show the effectiveness of the analysis using the PHARAON toolset to find good parallelization opportunities.

OpenStream parallelizations followed similar patterns (data-parallel loop and parallel sections) since no inter-task dependencies with streams looked promising. However, unlike the OpenMP-based parallelization, OpenStream focused on lower granularity parts of the code to leverage the efficiency of its run-time. The results obtained are reported in Table 2.

Table 1: Speedup for the stereo vision application on a 2 Intel i5 cores architecture using OpenMP.

Serial [s]	Parallel [s]	Speedup [times]
3.4	1.9	≈ 1.79

Table 2: Speedup for the stereo vision application on an 8 cores Intel architecture using OpenStream.

Serial [s]	Parallel [s]	Speedup [times]
4.06	1.16	≈ 3.5

4. CONCLUSION

This work presents the toolset and techniques developed in the PHARAON project with particular emphasis on the support for parallelization of legacy C code for multiprocessors platforms. These implement a complete flow, from UML modeling to final implementation, helping to reduce the development time, to increase the performance and to reduce the energy consumption.

The parallelization flow includes several tools. A performance estimation tool (Pareon) is used to extract timing and energy estimations for the code under analysis. The parallelization tool (ParTools) performs execution profiling and collects data dependencies program-wide at run-time. These, along with performance estimations, are shown in an interactive analysis interface at selectable levels of abstraction and analysis to help the developer decide on the best parallelization techniques and opportunities. The support for streaming-oriented parallelization is provided by OpenStream, an extension to the OpenMP standard.

The effectiveness of the parallelization toolset is demonstrated on two practical cases. One is a use case, involving unexperienced users, demonstrates the increment in parallelization quality and reduction of parallelization time due to the use of the toolset. The other demonstrates the use of the toolset for the parallelization of a stereo vision application of practical interest. The toolset helps to identify good parallelization candidates, at the proper level, and analyze their data dependencies and execution timings to define the best parallelization technique to achieve a significant speedup.

5. ACKNOWLEDGMENTS

This work is performed in the framework of the FP7-288307 funded project PHARAON. The stereo vision application described in this paper has been kindly provided by Tedesys within the PHARAON project.

6. ADDITIONAL AUTHORS

Alexandru Sutii (Vector Fabrics, Eindhoven, The Netherlands, email: alexandru@vectorfabrics.com).

7. REFERENCES

[1] T. Ahonen, T. D. Braak, S. T. Burgess, R. Geißler, P. M. Heysters, H. Hurskainen, H. G. Kerkhoff, A. B. Kokkeler, J. Nurmi, J. Raasakka, G. K. Rauwerda, G. J. Smit, K. Sunesen, H. Zonneveld, B. Vermeulen,

and X. Zhang. CRISP: Cutting Edge Reconfigurable ICs for Stream Processing. In J. M. P. Cardoso and M. Hübner, editors, *Reconfigurable Computing*, pages 211–237. Springer New York, 2011.

[2] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick. A View of the Parallel Computing Landscape. *Communications of the ACM*, 52(10):56–67, Oct. 2009.

[3] E. Athanasaki, N. Anastopoulos, K. Kourtis, and N. Koziris. Exploring the performance limits of simultaneous multithreading for memory intensive applications. *The Journal of Supercomputing*, 44(1):64–97, 2008.

[4] S. Balacco and C. Rommel. Next Generation Embedded Hardware Architectures: Driving Onset of Project Delays, Costs Overruns, and Software Development Challenges. Technical report, VDC Research Group, Inc., Sept. 2010.

[5] J. Barba, F. Rincón, F. Moya, J. D. Dondo, and J. C. López. A comprehensive integration infrastructure for embedded system design. *Microprocessors and Microsystems*, 36(5):383–392, 2012. Special Issue on Design of Circuits and Integrated Systems.

[6] M.-W. Benabderrahmane, L.-N. Pouchet, A. Cohen, and C. Bastoul. The Polyhedral Model Is More Widely Applicable Than You Think. In R. Gupta, editor, *Compiler Construction*, volume 6011 of *Lecture Notes in Computer Science*, pages 283–303. Springer Berlin Heidelberg, 2010.

[7] J.-J. Chen, C.-Y. Yang, T.-W. Kuo, and C.-S. Shih. Energy-Efficient Real-Time Task Scheduling in Multiprocessor DVS Systems. In *Proceedings of the 2007 Asia and South Pacific Design Automation Conference*, ASP-DAC '07, pages 342–349, Washington, DC, USA, 2007. IEEE Computer Society.

[8] B. Goossens and D. Parello. Limits of Instruction-Level Parallelism Capture. *Procedia Computer Science*, 18(0):1664–1673, 2013. <ce:title>2013 International Conference on Computational Science</ce:title>.

[9] J. L. Hennessy and D. A. Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2012.

[10] W.-M. Hwu, K. Keutzer, and T. Mattson. The concurrency challenge. *Design Test of Computers, IEEE*, 25(4):312–320, 2008.

[11] C. Isci, A. Buyuktosunoglu, C.-Y. Cher, P. Bose, and M. Martonosi. An Analysis of Efficient Multi-Core Global Power Management Policies: Maximizing Performance for a Given Power Budget. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 347–358, Washington, DC, USA, 2006. IEEE Computer Society.

[12] S. Kang, H. Kim, J. Baik, H. Choi, and C. Keum. Transformation Rules for Synthesis of UML Activity Diagram from Scenario-Based Specification. In *Computer Software and Applications Conference*, pages 431–436, July 2010.

[13] V. Kathail, S. Aditya, R. Schreiber, B. Rau, D. Cronquist, and M. Sivaraman. PICO: automatically designing custom computers. *Computer*, 35(9):39–47, Sept. 2002.

[14] B. Kienhuis, E. Rijpkema, and E. F. Deprettere. Compaan: deriving process networks from matlab for embedded signal processing architectures. In *Proceedings of the Eighth International Workshop on Hardware/Software Codesign*, pages 13–17, 2000.

[15] L. Lavagno, M. T. Lazarescu, I. Papaefstathiou, A. Brokalakis, J. Walters, B. Kienhuis, and F. Schäfer. HEAP: A Highly Efficient Adaptive multi-Processor framework. *Microprocessors and Microsystems*, 37(8, Part C):1050–1062, 2013. Special Issue on European Projects in Embedded System Design: {EPESD2012}.

[16] M. Lazarescu and L. Lavagno. Dynamic Trace-Based Data Dependency Analysis for Parallelization of C Programs. In *Source Code Analysis and Manipulation*, pages 126–131, Sept. 2012.

[17] N. M. Lê, A. Pop, A. Cohen, and F. Zappa Nardelli. Correct and Efficient Work-stealing for Weak Memory Models. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 69–80, New York, NY, USA, 2013. ACM.

[18] P. P. C. Lee, T. Bu, and G. Chandranmenon. A Lock-free, Cache-efficient Shared Ring Buffer for Multi-core Architectures. In *Proceedings of the 5th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '09, pages 78–79, New York, NY, USA, 2009. ACM.

[19] W. Mueller and Y. Vanderperren. UML and model-driven development for SoC design. In *Hardware/Software Codesign and System Synthesis*, pages 1–1, Oct. 2006.

[20] G. Ottoni, R. Rangan, A. Stoler, and D. August. Automatic thread extraction with decoupled software pipelining. In *Microarchitecture*, 2005.

[21] A. Pop and A. Cohen. OpenStream: Expressiveness and Data-flow Compilation of OpenMP Streaming Programs. *ACM Trans. Archit. Code Optim.*, 9(4):53:1–53:25, Jan. 2013.

[22] A. Prasad, J. Anantpur, and R. Govindarajan. Automatic Compilation of MATLAB Programs for Synergistic Execution on Heterogeneous Processors. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 152–163, New York, NY, USA, 2011. ACM.

[23] D. Shin and J. Kim. Power-aware Scheduling of Conditional Task Graphs in Real-time Multiprocessor Systems. In *Proceedings of the 2003 International Symposium on Low Power Electronics and Design*, ISLPED '03, pages 408–413, New York, NY, USA, 2003. ACM.

[24] F. Thoma, M. Kuhnle, P. Bonnot, E. Panainte, K. Bertels, S. Goller, A. Schneider, S. Guyetant, E. Schuler, K. Muüller-Glaser, and J. Becker. Morpheus: Heterogeneous reconfigurable computing. In *Field Programmable Logic and Applications*, pages 409–414, Aug 2007.