Speedups from Executing Critical Software Segments to Coarse-Grain Reconfigurable Logic

MICHALIS D. GALANIS¹, GREGORY DIMITROULAKOS², COSTAS E. GOUTIS³
VLSI Design Laboratory, Electrical & Computer Eng. Dept., University of Patras, Greece

Abstract: - In this paper, we propose a hardware/software partitioning method for improving applications’ performance in embedded systems. Critical software parts are accelerated on hardware of a single-chip generic system comprised by an embedded processor and coarse-grain reconfigurable hardware. The reconfigurable hardware is realized by a 2-Dimensional array of Processing Elements. A list-based mapping algorithm has been developed for estimating the execution cycles of kernels on Coarse-Grain Reconfigurable Arrays. The proposed partitioning flow has been largely automated for a program description in C language. Extensive hardware/software experiments on five real-life applications are presented. The results illustrate that by mapping critical code on coarse-grain reconfigurable hardware, speedups ranging from 1.2 to 3.7, with an average value of 2.2, are achieved.

Key-Words: - Reconfigurable embedded systems, Coarse-grain reconfigurable array, performance improvement, partitioning, kernel identification, mapping.

1. Introduction

Hardware/software partitioning is the procedure of dividing an application into software executed on microprocessor(s) and in custom-made co-processor units [1]. Partitioning can improve performance [1], and in some cases even reduce power consumption. More recently, hardware/software partitioning techniques for System-on-Chip (SoC) platforms composed by a microprocessor and FPGA [2], [3], were developed. The FPGA unit is treated as an extension of the microprocessor. Critical parts of the application, called kernels, are moved for execution on the FPGA for improved performance and usually reduced energy consumption. So, an extensive solution space search, as in past hardware/partitioning works [1] is not necessary.

Speedups are greater when coarse-grain reconfigurable hardware is used for accelerating DSP and multimedia applications instead on FPGA (fine-grain reconfigurable) hardware [4], [5]. Coarse-grain reconfigurable architectures have been proposed for accelerating loops of multimedia and DSP applications in embedded systems. They consist of a large number of Processing Elements (PEs) connected with a reconfigurable interconnect network. This work considers coarse-grain architectures where the PEs are organized in a 2-Dimensional (2D) array and they are connected with mesh-like reconfigurable networks [4], [6], [7]. This type of reconfigurable architecture is increasingly gaining interest because it is simple to be constructed and it can be scaled up, since more PEs can be added in the mesh-like interconnect. In this paper, these architectures are called Coarse-Grain Reconfigurable Arrays (CGRAs).

A hardware/software partitioning flow for accelerating critical software parts on the coarse-grain reconfigurable hardware is proposed in this work. The CGRA is coupled with an embedded processor in a SoC platform. The processor executes the non-critical software parts. This type of partitioning is possible in embedded systems, where the application is usually invariant during the lifetime of the system. Such systems are present in both academia [6], [7], and in industry, [8], [9]. These SoCs is expected to further gain importance since the coarse granularity of CGRAs greatly reduces the execution time and power consumption of critical software parts relative to an FPGA device at the expense of flexibility [4]. Thus, a formalized partitioning methodology like the one presented in this paper, is considered as a prerequisite for improving the performance of applications in such embedded systems.

A partitioning approach for a processor-CGRA architecture is presented in [10]. That work targeted programs described in subset of C language and it was applied in an image smoothing algorithm. A hardware/software partitioning approach for a system composed by a processor and a CGRA was presented in [9]. A speedup of an 8x8 IDCT by a factor of 4, relative to an all-software implementation, is reported. In [9], [10], the partitioning approaches were evaluated for one relatively small program. On the contrary, in this work we provide extensive results for five real-life DSP applications coded in C language: an IEEE
Orthogonal Frequency Division Multiplexing (OFDM) transmitter, a video compression technique, a medical imaging application, a wavelet-based image compressor and a JPEG compliant image encoder. The experiments show that the applications’ kernels contribute an average of 68.5% of the total dynamic instruction count, while their size is 10.9% on average of the total code size. Additionally, by mapping kernels on a 4x4 CGRA, speedups ranging from 1.2 to 3.7, with an average value of 2.2, are achieved relative to an all-software solution. To our knowledge, the presented experiments represent the most analytical hardware/software partitioning approach for processor-CGRA SoCs up to now.

The rest of the paper is organized as follows: section 2 presents the partitioning method, while section 3 describes the kernels’ identification flow. Section 4 describes the CGRA architecture template and the mapping algorithm for it. Section 5 presents the experimental results and section 6 concludes this paper.

2. Partitioning method

2.1. System architecture

A general diagram of the considered hybrid SoC architecture, that targets embedded applications, is shown in Fig. 1. The platform includes: (a) coarse-grain reconfigurable hardware for executing kernels, (b) shared system data memory, (c) instruction and configuration memories, and (d) an embedded microprocessor. The coarse-grain hardware is a CGRA. The microprocessor is typically a RISC processor, like an ARM7. Communication between the CGRA and the microprocessor takes place via the system’s shared data memory. Direct communication is also present between the CGRA and the processor. Part of the direct signals is used by the processor for controlling the CGRA by writing values to configuration registers located in the CGRA. The rest direct signals are used by the CGRA for informing the processor. Local data and configuration memory exist in the CGRA, for quickly loading data and configurations.

![Fig. 1. Considered hybrid SoC.](image)

2.2. Method description

The proposed hardware/software partitioning method for processor-CGRA systems interests in increasing application’s performance by mapping critical software parts on the coarse-grain reconfigurable hardware. The flow is illustrated in Fig. 2. The input is a software description of the application in a high-level language, like ANSI C/C++. Firstly, the Control Data Flow Graph (CDFG) Intermediate Representation (IR) is created from the input source code. The CDFG is the input to the kernel identification step. In the kernel identification, an ordering of the basic blocks in terms of the computational complexity is performed. The computational complexity is represented by the instruction count, which is the number of instruction executed in running the application on the microprocessor. The dynamic instruction count has been also used as a measure for identifying critical loop structures in previous works [2]. However, in this work the computational complexity is defined at a smaller granularity, at the basic block level. The instruction count is found by a combination of dynamic (profiling) and static analysis. A threshold, set by the designer, is used to characterize specific basic blocks as kernels. The rest of the basic blocks, that represent the non-critical application’s part, are going to be executed on the processor.

![Fig. 2. Hardware/software partitioning method.](image)

The kernels are mapped on the CGRA architecture by utilizing our-developed algorithm presented in section 4. The execution cycles are reported from this mapping procedure. The non-critical application’s parts are converted from the CDFG IR back to the source code representation. Then, the source code is compiled using a compiler for the specific processor and it is run on the microprocessor. The separation of the application to critical and non-critical parts, defines the data communication requirements between the processor and the CGRA. The proposed design flow considers the data exchange time through the shared memory for calculating the application’s execution time.
Currently, we consider the case where the processor and the CGRA execute in mutual exclusion. The kernels are replaced in the software with function calls to CGRA. When a call to CGRA is reached in the software, the processor activates the CGRA and the proper configuration is loaded on the CGRA for executing the kernel. When the CGRA executes a specific critical software part, the processor remains idle. After the completion of the kernel execution, the CGRA informs the processor, by typically using a direct interrupt signal, and writes the data required for executing the remaining software. Then, the execution of the software is continued on the processor and the CGRA remains idle.

The hardware/software partitioning method has been largely automated for a software description in C language. In particular, for the CDFG creation from the C code, we have used the SUIF2 and MachineSUIF compiler infrastructures. The automation of the kernel detection step is analytically described in section 3. The mapping algorithm for the CGRA, which is presented in section 4, is implemented in C++. For translating from the CDFG format to the C source code, the m2c compiler pass from the MachineSUIF distribution is utilized.

3. Kernel identification

The kernel identification step of the partitioning method outputs the kernel and non-critical parts of the software description. The inherent computational complexity of basic blocks, represented by the dynamic instruction count, is a meaningful measure to detect kernels. The instruction count when an application runs on the microprocessor is obtained by a combination of profiling and static analysis within basic blocks. Fig. 3 shows the kernel identification flow.

The input to the kernel detection process is the CDFG IR of the input source code. For the CDFG representation, we have chosen the SUIF Virtual Machine (SUIFvm) representation for the instruction opcodes inside basic blocks. The SUIFvm instruction set assumes a generic RISC machine, not biased to any existing architecture. Thus, the information obtained from the kernel identification flow, could stand for any RISC processor architecture. This means that the detected critical software parts are kernels for various types of RISC processors. This was justified by using the profiling utilities of the compilation tools of the processors considered in the experiments. In fact, the order of the instruction counts of the basic blocks is retained in the RISC processors used in our experiments.

![Fig. 3. Kernel identification diagram.](image)

We have used the MachineSUIF distribution for performing profiling at the basic block level. The profiling step reports the execution frequency of the basic blocks. For the static analysis, a MachineSUIF compiler pass has been developed that identifies the type of instructions inside each basic block. Afterwards, another developed pass calculates the static size of the basic block using the SUIFvm opcodes. The static size and the execution frequency of the basic blocks are inputs to a developed instruction mix pass that outputs the dynamic instruction count. After the instruction count calculation for each basic block, an ordering of the basic blocks is performed. We consider kernels, the basic blocks which have an instruction count over a user-defined threshold. This threshold represents the percentage of the contribution of the basic block’s instruction count in the application’s overall dynamic instructions.

4. CGRA mapping algorithm

4.1. Architecture template

The considered generic CGRA template is based on characteristics found in the majority of existing 2D coarse-grain reconfigurable architectures [4], [6], [7], and it can be used as a realistic model for mapping applications to such type of architectures. The proposed architecture template is shown in Fig. 4. Each PE is connected to its nearest neighbours, while there are cases [7] where there are also direct connections among all the PEs across a column and a row. A PE typically contains one Functional Unit (FU), which it can be configured to perform a specific word-level operation, each time. Characteristic operations supported by the FU are: ALU, multiplication, and shifts. For storing intermediate values between computations and data
fetched from memory, a small local RAM exists inside a PE.

The configuration memory of the CGRA (Fig. 4a) stores the whole configuration for setting up the CGRA for the execution of application’s kernels. Configuration caches distributed in the CGRA and reconfiguration (context) registers inside the PEs are used for the fast reconfiguration of the CGRA. The configuration cache stores a few contexts locally, which can be loaded on cycle-by-cycle basis. This reconfiguration mechanism can support one cycle reconfiguration of the CGRA. The CGRA’s data memory interface (Fig. 4b) consists of: (a) the memory buses, (b) the scratch-pad memory which is the first level L1 of the CGRA’s memory hierarchy, and (c) the base memory level, called L0, which is formed by the local RAMs inside the PEs. The main data memory of the CGRA is a part of the system’s shared data memory (Fig. 1). As it happens in the majority of the existing CGRA architectures [6], [7], the PEs residing in a row or column share a common bus connection to the scratch-pad memory. The L1 serves as a local memory for quickly loading data in the PEs of the CGRA. The interconnection network together with the L0 acts as a high-bandwidth foreground memory, since during each cycle several data transfers can take place through different paths in the CGRA.

Fig. 4. (a) CGRA architecture template, (b) CGRA’s data memory hierarchy.

4.2. Algorithm description

The first input to the mapping algorithm is a Data Flow Graph (DFG) $G(V, E)$ that represents the kernel which is to be mapped to the CGRA. The algorithm is applied to all the application’s kernels for computing the execution cycles on the CGRA. The description of the CGRA architecture is the second input to the mapping phase. The CGRA architecture is modelled by a undirected graph, called CGRA Graph, $G_{N}(V_p, E_i)$. The $V_p$ is the set of PEs of the CGRA and $E_i$ are the interconnections among them. The CGRA architecture description includes parameters, like the size of the local RAM inside a PE, the memory buses to which each PE is connected, the bus bandwidth and the memory access times.

The PE selection for scheduling an operation, and the way the input operands are fetched to the specific PE, will be referred to hereafter as a Place Decision (PD) for that specific operation. Each PD has a different impact on the operation’s execution time and on the execution of future scheduled operations. For this reason, a cost is assigned to each PD to incorporate the factors that influence the scheduling of the operations. The goal of the mapping algorithm is to find a cost-effective PD for each operation. The proposed mapping algorithm is shown in Fig. 5 and it is a list-based one. The algorithm is initialized by assigning to each DFG node a value that represents its priority. The priority of an operation is calculated as the difference of its As Late As Possible (ALAP) minus its As Soon As Possible (ASAP) value. This result is called mobility. Also variable $p$, which indirectly points each time to the most exigent operations, is initialized by the minimum value of mobility. In this way, operations residing in the critical path are considered first in the scheduling phase. During the scheduling phase, in each iteration of the while loop, QOP queue takes via the $ROP()$ function the ready to be executed operations which have a value of mobility less than or equal to the value of variable $p$. The first do-while loop schedules and routes each operation contained in the QOP queue one at a time, until it becomes empty. Then, the new ready to be executed operations are considered via $ROP()$ function which updates the QOP queue.

![Fig. 5. Mapping algorithm for CGRAs.](image-url)
The Predecessors() function returns (if exist) the PEs where the Op’s predecessors (Pred_PEs) were scheduled and the earliest time (RTime) at which the operation Op can be scheduled. The RTime equals to the maximum of the times, where each of the Op’s predecessors finished executing. The function GetCosts() returns the possible PDs and the corresponding costs for the operation Op in the CGRA, in terms of the Choices variable. It takes as inputs the earliest possible schedule time (RTime) for the operation Op along with the PEs where the Pred_PEs have been scheduled. The function ResourceCongestion() returns true if there are no available PDs due to resources constraints. In that case RTime is incremented and the GetCosts() function is repeated until available PDs are found. The DecideWhereToScheduleTimePlace() function analyzes the mapping costs from the Choices variable. The function firstly identifies the subset of PDs with minimum delay cost. From the resulting PD subset, it selects the one with minimum interconnection cost as the one which will be adopted. The function ReserveResources() reserves the resources (bus, PEs, local RAMs and interconnections) for executing the current operation on the selected PE. More specifically the PEs are reserved as long as the execution takes place. Finally, the Schedule() records the scheduling of operation Op. After all operations are scheduled, the execution cycles are reported.

5. Experimental results
5.1. Set-up
We have used five applications, written in C language, for demonstrating the effectiveness of the partitioning flow. The first one is the baseband processing of an IEEE 802.11a OFDM transmitter. The second application is a cavity detector which is a medical image processing application. The third one is a video compression technique, called Quadtree Structured Difference Pulse Code Modulation (QSDPCM), while the fourth one is a still-image JPEG encoder. Finally, the fifth application is a wavelet-based image compressor.

We have used four different types of 32-bit embedded RISC processors: an ARM7, an ARM9, and two SimpleScalar processors. The SimpleScalar processor is an extension of the MIPS32 IV core. These processors are widely used in embedded SoCs. The first type of the MIPS processor (MIPSa) uses one integer ALU unit, while the second one (MIPSb) has two integer ALU units. We have used instruction-set simulators for the considered embedded processors for estimating the number of execution cycles. More specifically, for the ARM processors, the ARM Developer Suite (version 1.2) was utilized, while the performance for the MIPS-based processors is estimated using the SimpleScalar simulator tool. Typical clock frequencies are considered for the four processors: the ARM7 runs at 100 MHz, the ARM9 at 250 MHz, and the MIPS processors at 200 MHz.

The CGRA architecture used in this experimentation is a 4x4 array of PEs. The PEs are directly connected to all other PEs in the same row and same column through vertical and horizontal interconnections, as in a quadrant of MorphoSys [7]. There is one 16-bit FU in each PE that can execute any operation in one CGRA’s clock cycle. Each PE has a local RAM of size 64 words; thus the L0 size is 2 Kbytes. The direct connection delay among the PEs is zero cycles. Two buses per row are dedicated for transferring data to the PEs from the scratch-pad (L1) memory. The delay of fetching one word from the scratch-pad memory is one cycle. Also, cycle-by-cycle reconfiguration of the CGRA is supported. The CGRA’s clock frequency is 100 MHz, as in the case of an implementation of the MorphoSys SoC [7].

5.2. Experimentation
The performance results from applying the partitioning flow in the five applications are presented in Table 1. For each application, the four considered processor architectures (Proc. Arch.) are used for estimating the clock cycles (Cycles_{hw/sw}) required from executing the whole application on the processor. We have assumed two cases for mapping the kernels on the CGRA: (a) mapping of the original kernel body, and (b) unrolling the body 16 times. The ideal speedup (Ideal sp.) reports the maximum performance improvement, according to Amdahl’s Law, if application’s kernels were ideally executed on the CGRA in zero time. The estimated speedup (Est. Sp.) is the measured performance improvement after utilizing the developed partitioning method. The estimated speedup is calculated as:

\[
\text{Est. Sp.} = \frac{\text{Cycles}_{\text{init}}}{\text{Cycles}_{\text{hw/sw}}} \quad (1)
\]

where Cycles_{hw/sw} represents the execution cycles after the partitioning.

From the results given in Table 1, it is evident that significant performance improvements are achieved when critical software parts are mapped on a CGRA. It is noticed that better performance gains are achieved for the ARM7 case than the ARM9-CGRA scenario. This occurs since the speedup of kernels on the CGRA has greater effect when the CGRA is coupled with a lower-performance processor, as it is the ARM7 relative to the ARM9.
Furthermore, the speedup is almost always greater for the MIPSa case than the MIPSb case, since the latter one employs one more integer ALU unit.

For the case of unrolling the kernel’s bodies when mapped on the CGRA, the speedups are larger than the original bodies had enough instruction parallelism that was efficiently exploited by the mapping algorithm for the CGRA. The average estimated speedup is 2.0 for unroll factor equal to 1, while for unroll factor equal to 16 the speedup is 2.3. We also notice that the reported estimated speedups for each application and for each processor type are somewhat close to the ideal speedups determined by the Amdahl’s Law, especially for the case of unroll factor equal to 16. We have also performed experiments with a larger clock frequency of the CGRA, which was set to 150 MHz, equal to the frequency of the CGRA in [8]. In this case, the average speedup is 2.2 for unroll factor equal to 1, and 2.4 for unroll factor 16. The results of this section prove the effectiveness of the proposed hardware/software partitioning method.

### 6. Conclusions

A flow for accelerating critical software segments in processor-CGRA SoCs was presented. Important performance improvements have been achieved. The speedup ranges from 1.2 to 3.7 when a 4x4 CGRA is used for kernels’ acceleration.

### Table 1. Execution cycles and speedups for the processor-CGRA SoC

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cavity</td>
<td>ARM7</td>
<td>178,828,950</td>
<td>2.4</td>
<td>94,114,190</td>
<td>1.9</td>
<td>88,020,014</td>
<td>2.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ARM9</td>
<td>161,441,889</td>
<td>2.3</td>
<td>102,071,601</td>
<td>1.6</td>
<td>86,836,161</td>
<td>1.9</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MIPSa</td>
<td>470,433,835</td>
<td>2.3</td>
<td>255,064,454</td>
<td>1.8</td>
<td>242,876,102</td>
<td>1.9</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MIPSb</td>
<td>310,248,110</td>
<td>2.2</td>
<td>178,446,531</td>
<td>1.7</td>
<td>166,258,179</td>
<td>1.9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OFDM</td>
<td>ARM7</td>
<td>397,851</td>
<td>3.3</td>
<td>141,819</td>
<td>2.8</td>
<td>128,998</td>
<td>3.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ARM9</td>
<td>362,990</td>
<td>3.4</td>
<td>153,662</td>
<td>2.4</td>
<td>121,584</td>
<td>3.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MIPSa</td>
<td>459,594</td>
<td>3.4</td>
<td>173,683</td>
<td>2.7</td>
<td>148,021</td>
<td>3.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MIPSb</td>
<td>352,788</td>
<td>3.3</td>
<td>147,500</td>
<td>2.4</td>
<td>121,838</td>
<td>2.9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wavlet</td>
<td>ARM7</td>
<td>25,832,508</td>
<td>2.5</td>
<td>13,347,644</td>
<td>1.9</td>
<td>11,700,124</td>
<td>2.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ARM9</td>
<td>20,574,658</td>
<td>2.3</td>
<td>14,952,962</td>
<td>1.4</td>
<td>10,474,162</td>
<td>2.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MIPSa</td>
<td>62,468,206</td>
<td>2.5</td>
<td>31,415,355</td>
<td>2.0</td>
<td>28,120,315</td>
<td>2.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MIPSb</td>
<td>40,541,866</td>
<td>2.3</td>
<td>23,121,852</td>
<td>1.8</td>
<td>19,826,812</td>
<td>2.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JPEG</td>
<td>ARM7</td>
<td>23,003,868</td>
<td>4.0</td>
<td>7,068,598</td>
<td>3.3</td>
<td>6,268,096</td>
<td>3.7</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ARM9</td>
<td>19,951,193</td>
<td>3.2</td>
<td>8,926,634</td>
<td>2.6</td>
<td>6,925,379</td>
<td>2.9</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MIPSa</td>
<td>34,451,609</td>
<td>3.3</td>
<td>13,054,733</td>
<td>2.6</td>
<td>11,453,728</td>
<td>3.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MIPSb</td>
<td>19,637,417</td>
<td>3.2</td>
<td>8,395,362</td>
<td>2.3</td>
<td>6,794,357</td>
<td>2.9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>QSDPCM</td>
<td>ARM7</td>
<td>4,026,384,618</td>
<td>1.6</td>
<td>3,161,212,458</td>
<td>1.3</td>
<td>3,079,646,772</td>
<td>1.7</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ARM9</td>
<td>3,895,248,922</td>
<td>1.5</td>
<td>3,253,991,290</td>
<td>1.2</td>
<td>3,050,077,075</td>
<td>1.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MIPSa</td>
<td>7,006,016,541</td>
<td>1.7</td>
<td>4,859,004,618</td>
<td>1.4</td>
<td>4,695,873,259</td>
<td>1.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MIPSb</td>
<td>4,910,759,258</td>
<td>1.7</td>
<td>3,584,022,810</td>
<td>1.4</td>
<td>3,420,891,438</td>
<td>1.4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Average:** 2.0 2.3

### References


