Hardware Pipelining of Repetitive Patterns in Processor Instruction Traces

João Bispo[1], João M. P. Cardoso[2] and José Monteiro[3]

[1,3] CSE Dept., IST/UTL, INESC-ID, Lisboa, Portugal
[2] INESC-TEC and Dept. of Inf. Eng., FEUP/Univ. of Porto, Porto, Portugal
e-mail: joaobispo@gmail.com, jmpc@acm.org, jcm@inesc-id.pt

ABSTRACT

Dynamic partitioning is a promising technique where computations are transparently moved from a General Purpose Processor (GPP) to a coprocessor during application execution. To be effective, the mapping of computations to the coprocessor needs to consider aggressive optimizations. One of the mapping optimizations is loop pipelining, a technique extensively studied and known to allow substantial performance improvements. This paper describes a technique for pipelining Megablocks, a type of runtime loop developed for dynamic partitioning. The technique transforms the body of Mega-blocks into an acyclic dataflow graph which can be fully pipe-lined and is based on the atomic execution of loop iterations. For a set of 9 benchmarks without memory operations, we generated pipelined hardware versions of the loops and esti-mate that the presented loop pipelining technique increases the average speedup of non-pipelined coprocessor accelerated designs from 1.6× to 2.2×. For a larger set of 61 benchmarks which include memory operations, we estimate through simulation a speedup increase from 2.5× to 5.6× with this technique.

Index Terms: Reconfigurable Fabrics; Dynamic Mapping; Hardware Acceleration; Instruction Traces; Loop Pipelining.

I. INTRODUCTION

The performance of application running on a general purpose processor (GPP) can be enhanced by moving computationally intensive parts (hotspots) to a Coprocessor Unit (CU, or simply coprocessor) [1-3]. While many different approaches can be used to couple these two components [4], in a common approach (illustrated in Fig. 1) the CU communicates with the GPP by direct connections and both have access to the system memory (i.e., the CU acts as a traditional coprocessor).

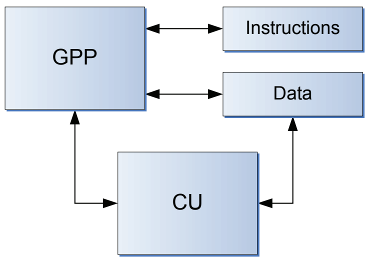

Figure 1: Block diagram of a typical target system which includes a coprocessor (CU) acting as an accelerator of the GPP.

Hardware/software co-design [5, 6] is a methodology for designing embedded systems consisting of hardware and software components. An important part of this methodology is the identification and mapping of application hotspots to hardware (referred herein as hardware/software partitioning, or simply partitioning). Partitioning contains steps such as the detection of computation-intensive sections in the application (also known as hotspots or critical sections), mapping the computations to each of the components of the target architecture (i.e., the software and the hardware components), and adapting the software application to use the hardware component (e.g., calls to custom hardware units are inserted in the application source code). This usually requires the insertion of synchronization and data communication primitives.

Most efforts perform partitioning statically and require the source code of the application (in some contexts not available). Using the traditional approach, the final implementation is crystallized and is not adapted to different runtime characteristics.

Dynamic partitioning and mapping of computations (hereafter referred as dynamic partitioning) is a promising technique able to transparently move computations from a GPP to a coprocessor in a transparent way, and may become an important contribution for the future reconfigurable embedded computing systems. Dynamic partitioning can be of paramount importance in allowing applications to take advantage of reconfigurable fabrics existent in the host system and thus to increase performance portability.

To increase the efficiency of dynamic mapping techniques, one needs to consider optimizations, such as loop pipelining. When mapping loops to Coarse-Grained Reconfigurable Arrays (note that CGRAs can be considered one possible type of CU), performance can significantly improve if the iterations of the loop are pipelined [7]. However, its use in the context of dynamic partitioning has been neglected: it is usually considered a complex optimization.

This paper presents a novel technique for pipelining the iterations of Megablocks [8], a type of runtime loop specifically developed to be used in dynamic partitioning. By taking advantage of the characteristics of the Megablock, it was possible to develop a lightweight pipelining technique. Although being applied statically in the context of this paper, it is being developed bearing in mind its application at runtime. Additionally, the technique is able to commit loop iterations atomically, avoiding the implementation of an epilogue, and it proposes a module which handles certains memory dependencies.

The rest of this paper is organized as follows. Section II introduces the Megablock and Section III explains how Megablocks can be pipelined. Section IV proposes an architecture which implements pipelined Megablocks. Section V presents results for two sets of benchmarks, and Section VI discusses related work in this area. Finally, Section VII concludes the paper and presents some future work possibilities.

II. THE MEGABLOCK

A Megablock [8] is a runtime loop which continuously repeats the *same* sequence of instructions until an exit condition is activated. It represents a repetitive path formed during runtime, it has a single execution path and one or more exit points. Megablocks are by definition loops with well-defined control-flow, and can be extracted from loops with arbitrary static control-flow if, during execution, the loop behaves in such a way that forms patterns.

The Megablock formulation is as follows. Consider a program P, which is formed by the sequence of instructions $[i_1 i_2 \dots i_m]$. Each instruction i_j is uniquely identified by an address. The execution of P generates a sequence T, called a trace, formed by instructions from P. Consider S as an arbitrary size sequence of instructions one can find in T (e.g., $[i_5\, i_6\, i_7]$ and $[i_8\, i_2]$ are two specific instruction sequences). A Megablock is a sequence S, such that $S\{n\}$, with n greater than 1 and representing the number of times the sequence S repeats, forms a contiguous subsequence of T (e.g., consider a Megablock $S=[i_5\, i_6\, i_7]$. If $S\{3\}$ is found in T, it means that $[i_5\, i_6\, i_7\, i_5\, i_6\, i_7\, i_5\, i_6\, i_7]$ is a contiguous subsequence in T).

As an example, consider the *vecsum* function depicted in Fig. 2. When a GPP executes the loop in *vecsum*, a sequence of GPP instructions is repeatedly executed. Fig. 3 shows the repeating pattern of a Megablock found in the execution trace of *vecsum* when executed in a MicroBlaze processor [9]. The MicroBlaze instructions are translated to processor-independent instructions, which are then used to build a graph-based intermediate representation. Fig. 4 represents the same Megablock pattern as a graph.

In this graph representation, rounded nodes represent operations; transparent square nodes represent constants; and filled square nodes represent inputs and exits. Input nodes represent values which are provided by the GPP before loop execution begins. Updates to input nodes are represented by connections with the label *feedback*. Data connections between operations contain a label which follow the format OUT:IN, where OUT and IN correspond to the output and input index of the source and of the destination node, respectively.

```
void vecsum(int* A, int* B, int* C, int n) {
  int i;
  for(i = 0; i < n; i++) {
    C[i] = A[i] + B[i];
  }
}
```

Figure 2. C code for the vecsum function.

```
0x00000180 lw r3, r5, r9       → 0:add, 1:load
0x00000184 lw r4, r6, r9       → 2:add, 3:load
0x00000188 addik r10, r10, 1   → 4:add
0x0000018C addk r3, r3, r4     → 5:add
0x00000190 sw r3, r7, r9       → 6:add, 7:store
0x00000194 rsubk r18, r10, r8  → 8:rsub_carry
0x00000198 bneid r18, -24      → 9:equalZero
0x0000019C addik r9, r9, 4     → 10:add
```

Figure 3. MicroBlaze instructions of the pattern of a Megablock in the execution trace of vecsum and their translation to intermediate operations.

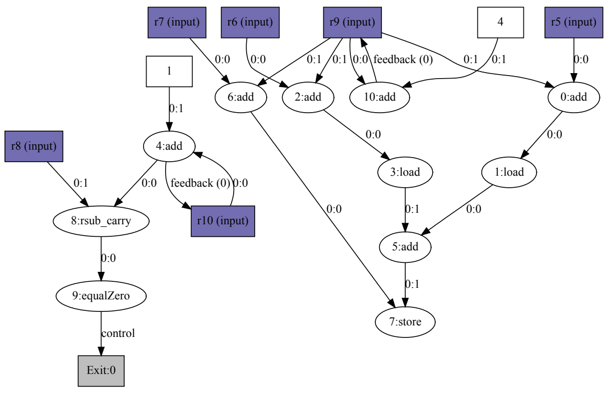

Figure 4. Graph representation of the Megablock pattern found in vecsum.

III. MEGABLOCK PIPELINING

When a loop is pipelined, subsequent iterations start executing before the previous iteration has finished. Data dependencies [10] between iterations (i.e., inter-iteration dependencies) can prevent the execution of iterations before certain conditions are met. When pipelining the iterations of a loop, it is necessary to guarantee that all true data dependencies are respected. We classify data dependencies in Megablocks into two categories: register and memory dependencies. Register dependencies are data dependencies between registers in assembly instructions and are explicitly represented in the Megablock graph by data connections. *Feedback* connections are data connections between consecutive iterations, and represent inter-iteration register dependencies. Memory dependencies are not explicitly represented, and correspond to operations which manipulate data in a medium external to the processor (e.g., memory accesses). To pipeline Megablocks, we propose a scheme capable of handling register dependencies, and that can be applied to loops without inter-iteration memory dependencies. This may seem a severe restriction, but does not prevent us to pipeline Megablocks found in many signal/image processing kernels as shown by the experimental results.

Currently, we consider that Megablocks without inter-iteration memory dependencies are manually identified by analysis of the source code. Automatic detection of the necessary conditions for pipelining generic Megablocks will be addressed in future work. This requires memory aliasing analysis for removing intra-iteration memory dependencies.

A. Inter-Iteration Register Dependencies in Megablocks

Consider the graph representation in Fig. 4 of the Megablock depicted in Fig. 3. We developed an algorithm to identify the expressions responsible to control the value of the inputs in subsequent iterations, and to build directed graph representations for those expressions (see Fig. 5). In each *input* node with a *feedback* connection in Fig. 4, traversing the graph in the opposite direction of the connection reaches the node that generates the input value for the next iteration. E.g., following the *feedback* connection in node *r9 (input)* the values of the input are given by the output of node *10:add*. This is the condition to start a new graph. As it is the first time the algorithm sees the node *10:add*, this node is added to the graph. As this node is an operation, the algorithm is called recursively to each of its parent nodes. All the inputs of node *10:add* are either of type *input* or *constant*, thus after they are added to the graph, the algorithm stops. The resulting graph represents the update expression for *r9 (input)*, which in this case is $r9 = r9 + 4$. The algorithm continues by considering the next input node with a feedback connection, *r10 (input)*, and repeats the process. As this is the last input with a feedback connection, there are no more expressions to identify, and we obtain the *InputsGraph* in Fig. 5.

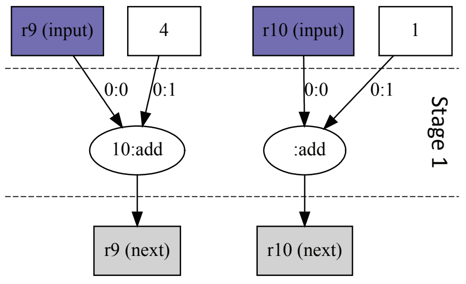

Figure 5. InputsGraph for the vecsum Megablock in Figure X.

The update of the values of the inputs through iterations can be handled by a hardware structure which implements the *InputsGraph*. This structure becomes responsible to feed a new set of values to the Megablock at the beginning of each iteration. This way, the *feedback* connections can be removed from the original Megablock graph (see Fig. 4), transforming it to an acyclic dataflow graph which can be fully pipelined. This technique is appropriate for loops where the operations related to the update of values used across iterations represent a small part of the loop. As we will see later, the lower the latency of the module related to the update of inputs, relative to the original loop, the greater the potential for improvements.

B. Inter-Iteration Memory Dependencies in Megablocks

A Megablock does not have inter-iteration memory dependencies if one can guarantee that: 1) store operations are done according to their original order; and 2) the contents of the addresses accessed by load operations are not changed during its execution. Guarantee 1) implies a mechanism for serializing memory writes, and can be enforced by hardware implementation. This satisfies output dependencies between memory writes. Guarantee 2) depends on the program and compiler options. Since with this guarantee the values accessed by load operations are immutable, it avoids true dependencies and anti-dependencies between memory accesses. This guarantee can be enforced when programs use separate memory areas (e.g., occurring with non -overlapped arrays) for reading and writing values. As Guarantee 1) can be enforced by hardware, we only have to ensure that Megablocks respect Guarantee 2). E.g., *vecsum* (see Fig. 2) uses different arrays for reading and writing, respecting Guarantee 2). Currently, we assume that information is provided by the compiler as additional information. Future work will address Megablock analysis to provide that information.

IV. ARCHITECTURE FOR PIPELINED MEGABLOCKS

Fig. 6 presents the modules needed by our approach for pipelining Megablocks. The solution in Fig. 6b) is a specialization of the solution in Fig. 6a), when considering Megablocks without memory accesses. Both solutions have an Input Module (IM) and a Loop Module (LM). The support for memory accesses (Fig. 6a) includes a Store Module (SM) and load units in the LM module. Both solutions aim at executing iterations atomically, i.e., iterations are either fully executed or discarded when an iteration activates an exit point.

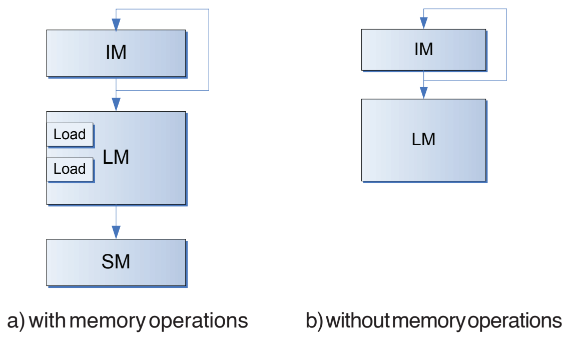

Figure 6. General block diagrams for pipelined execution of Megablocks.

The LM module represents a pipelined dataflow implementation of the Megablock repeating pattern and can be thought as the loop body split into several stages. Each preceding stage executes the next iteration of the Megablock, and when the LM advances a step, which can take one or more clock-cycles, depending on the Megablock and its implementation, all stages execute simultaneously. An iteration completes when it finishes execution at the last stage of the LM without activating exit points. All exit points are delayed so that when they are checked, the corresponding iteration is in the last stage. After filling the pipeline, the LM completes an iteration per step. To advance a step, the LM needs the values generated by the IM. The IM is responsible for generating the set of inputs per iteration, and only depends on the values generated in the previous step of the IM.

According to Guarantee 1) for memory dependencies, store operations have to be executed by their original order. Since the LM executes operations of different iterations simultaneously, the technique moves the store operations outside of the LM, i.e., to the SM. The LM delays all store operations until the last stage, and only executes them if no exits are activated for that iteration, avoiding speculative writes to memory. The SM depends on the results of the LM. According to Guarantee 2) for memory dependencies, load operations are done from immutable locations. This means that load operations can be done in any order, and remain in the LM. However, in this case the step of the LM module only finishes after all load operations complete.

Fig. 7 presents a possible schedule for the Loop Module of the Megablock graph in Fig. 4 when using an As-Soon-As-Possible (ASAP) based scheduler [11], after *feedback* connections are removed and store operations are moved to a separate module.

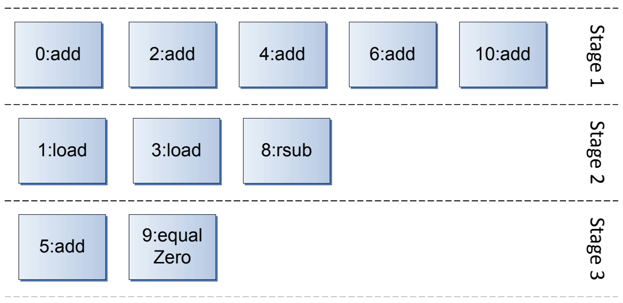

Figure 7. Loop Module (LM) schedule for a Megablock found in vecsum.

A. Pipeline Scheduling

One can execute the modules shown in Fig. 6 one at a time, sequentially (Fig. 8a). However, the IM only depends on its previous values, and as soon as it finishes execution, it can start computing the values of the next iteration. When pipelining the modules, we can overlap the execution of the IM with the remaining modules, leading to an *overlapping schedule* (Fig. 8b to d).

Consider that the IM execution is separated in two parts, IM-A and IM-L, which are executed concurrently. IM-A refers to the execution or arithmetic and logic operations (e.g., additions, subtractions). IM-L corresponds to the execution of load operations. In this model, store operations are not allowed in the IM. The IM is separated in these two components because the number of concurrent memory accesses usually is very limited in real systems, and when the IM execution overlaps with the execution of the remaining modules, they will compete for the same limited resources. We consider that the execution of the IM associated to load operations (IM-L) does not overlap with the remaining modules (LM and SM), which can also have memory operations.

The LM can have a similar decomposition, LM-A and LM-L, where the arithmetic and logic component executes concurrently with the both IM-A component and the memory related components, in a third overlapping level. However, as the LM is pipelined, the arithmetic-logic part usually executes within one clock cycle, and the load operations represent the longest execution part of the LM. For simplicity, this decomposition was not considered. When considering the case without memory accesses, we use a similar schedule, which includes neither the SM nor the decomposition.

Software pipelining algorithms [12, 13] usually have a prologue, a steady state, and an epilogue. The purpose of the epilogue is to orderly terminate the execution of iterations which cannot execute in the steady state because there are no more new iterations to feed the pipeline. Our approach does not have an epilogue. Since we atomically commit iterations that have finished, we can simply ignore the iterations which have already started but have not yet terminated by the time an exit activates. When an exit activates, the store instructions related to that iteration are not performed.

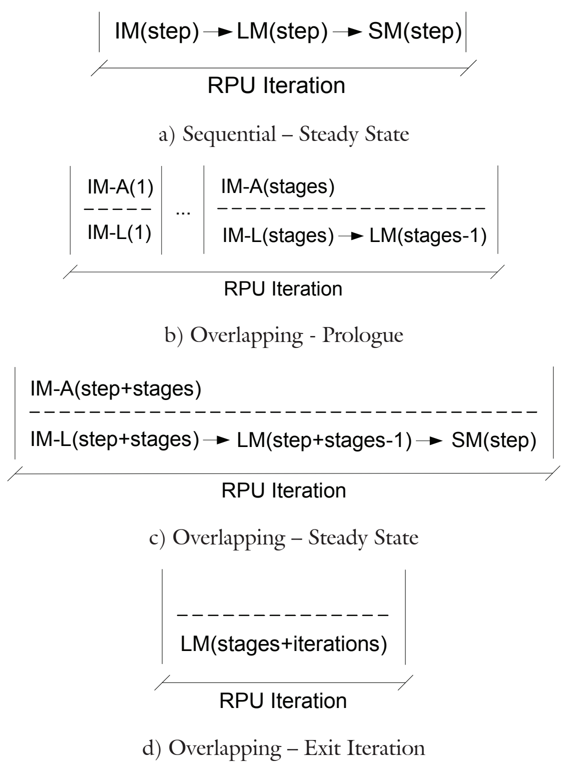
IM(step) → LM(step) → SM(step)
RPU Iteration
a) Sequential – Steady State
IM-A(1)
IM-L(1)
...
IM-A(stages)
IM-L(stages) → LM(stages-1)
RPU Iteration
b) Overlapping - Prologue
IM-A(step+stages)
IM-L(step+stages) → LM(step+stages-1) → SM(step)
RPU Iteration
c) Overlapping – Steady State
LM(stages+iterations)
RPU Iteration
d) Overlapping – Exit Iteration

Figure 8. Sequential and overlapping schedule for the modules of a pipelined RPU.

B. Calculating Coprocessor Latencies

In this section we describe analytical expressions to calculate estimations of the clock cycles needed to execute Megablocks. Term CU-xxx$_{Cy}$ represents the number of cycles of coprocessor execution for a specific case. Term LM$_{Stg}$ represents the number of LM stages, while N$_{It}$ represents the number of completed Megablock iterations.

Equations (1) and (2) define CU$_{Cy}$ for the overlapping schedule with memory accesses, while (3) considers the absence of memory accesses. The terms IM -A(i)$_{Cy}$, IM-L(i)$_{Cy}$, LM(i)$_{Cy}$, and SM(i)$_{Cy}$ (1) represent the clock cycles needed to complete the step i of the corresponding module. Equations (2) and (3) consider each module always execute in a fixed number of cycles, represented by the terms IM-A$_{Cy}$, IM-L$_{Cy}$, LM$_{Cy}$ and SM$_{Cy}$. Usually, the latency of the LM is determined by the latency of the load operations. Since the LM is pipelined, if an LM does not have load operations, it will have the shortest step between all modules (usually one clock cycle). In this case, the IM latency becomes the dominant term. Considering the overlapping schedule without memory accesses, this means that the *Max* operation in Equation (3) can in most cases be simplified to IM$_{Cy}$. Equation (4) represents the approximate number of clock cycles used by the coprocessor, for the overlapping schedule, when Megablocks execute for a large number of iterations, well above the number of stages of the LM. This equation is useful for calculating maximum theoretical speedup limits when comparing with non-pipelined versions of the architecture.

$$\begin{aligned}\text{CU-Mem-Var}_{Cy} &= Max(\text{IM-A(i)}_{Cy}, \text{IM-L(i)}_{Cy}) + \\ &\textstyle\sum_{i=2}^{LM_{Stg}} Max(\text{IM-A(i)}_{Cy}, \text{IM-L(i)}_{Cy} + \text{LM(i}-1)_{Cy}) + \\ &\textstyle\sum_{i=1}^{N_{It}} Max\left(\text{IM-A}(i+LM_{Stg})_{Cy}, \text{IM-L}(i+LM_{Stg})_{Cy}\right) + \\ &\text{LMi+LMStg}-\text{1Cy+SMiCy+LMNIt+LMStgCy}\end{aligned} \quad (1)$$

$$\begin{aligned}\text{CU-Mem-Fixed}_{Cy} &= Max(\text{IM-A}_{Cy}, \text{IM-L}_{Cy}) + \\ &Max(\text{IM-A}_{Cy}, \text{IM-L}_{Cy} + \text{LM}_{Cy}) \times (\text{LM}_{Stg} - 1) + \\ &Max(\text{IM-A}_{Cy}, \text{IM-L}_{Cy} + \text{LM}_{Cy} + \text{SM}_{Cy}) \times N_{It} + \text{LM}_{Cy}\end{aligned} \quad (2)$$

$$\begin{aligned}\text{CU-NoMem-Fixed}_{Cy} &= \text{IM-A}_{Cy} + Max(\text{IM-A}_{Cy}, \text{LM}_{Cy}) \times \\ &(\text{LM}_{Stg} + N_{It} - 1) + \text{LM}_{Cy}\end{aligned} \quad (3)$$

$$\begin{aligned}\text{CU-Overlapping}_{Cy} &= Max(\text{IM-A}_{Cy}, \text{IM-L}_{Cy} + \text{LM}_{Cy} + \\ &\text{SMCy} \times \text{NIt}\end{aligned} \quad (4)$$

C. Hardware Module for Pipelining Megablocks

Fig. 9 shows the general architecture for a 2D Unfolded CGRA, consisting of a reconfigurable array with several rows of FUs and forward communication links between rows (only the last row is represented in the figure). This CGRA contains an Iteration Control module that stops CGRA's execution if an exit condition is activated. The FUs which communicate with the Iteration Control module can be used to implement the operations which signal exits.

Note that this is just one of several possible architecture implementations. For instance, we can build non-reconfigurable hardware modules which implement a single Megablock using a similar architecture, or use an architecture that requires less resources (e.g., 1D Folded CGRA [14]). However, to take advantage of hardware loop pipelining, we need an architecture which physically implements the several stages of the pipeline.

To enable our Megablock pipelining approach in such CGRAs, we propose three hardware extensions presented in Fig. 10: (a) feedback lines to the top row, for the IM; (b) clock-enable control signals for each module; and (c) delays for the exit signals (for simplicity, the CGRA in the figure only has three rows). The extensions enable the implementation of the IM, the LM, and the SM at the hardware level. The *feedback* connections from intermediate rows to the top row (a) are needed for IM re-alimentation. This kind of interconnection can be expensive, but since only Input Modules with a low number of stages are attractive for implementation, these connections can be present in only a restricted number of top rows. Since the modules have producer-consumer relationships between them, we use a Step Controller (b) to send clock-enable control signals for each row of FUs. This way it is possible to indicate when there are values available for each module, and when they can proceed. The tapped delay lines (simple 1-bit shift-registers) for the exit signals (c) synchronize the signals so that when they activate, they always correspond to the iteration in the last stage.

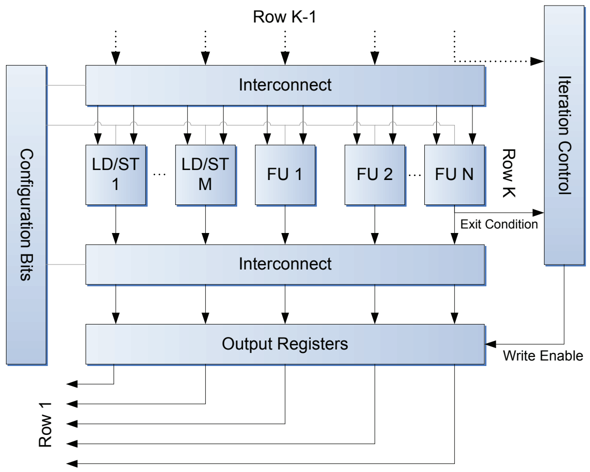

Figure 9. General architecture for a 2D CGRA-based RPU which supports Megablocks.

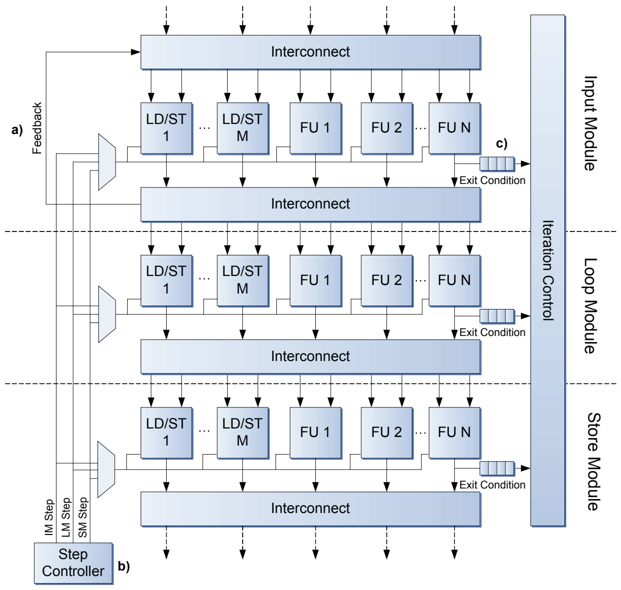

Figure 10. General architecture for a 2D CGRA-based coprocessor which supports Megablock pipelining.

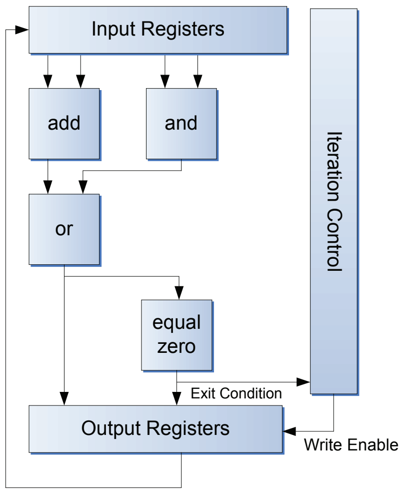

Figure 11. A specialized implementation for a possible Megablock.

If the mapping of Megablocks can be done at compile instead of runtime (e.g., with the help of profiling[15]), we can translate the Megablocks to HDL descriptions which directly implement the Megablocks (see Fig. 11). In this case we trade-off the flexibility of runtime mapping for specialized designs which can be more efficient resource and performance-wise. The execution is similar to what was described for the 2D Unfolded CGRA architecture in Fig. 9; the iterations are executed atomically, and the output of each operation is registered so that we can apply the same pipelining transformations.

V. EXPERIMENTAL RESULTS

We have developed a scheduler and a simulator tool for estimating the execution of a program when moving highly complex Megablocks to a coprocessor. The simulator considers the architecture presented in Fig. 10. To evaluate the proposed pipelining technique, we developed a proof-of-concept VHDL generator which converts a Megablock graph into a specialized hardware module. For each Megablock, the tool can generate a hardware module in VHDL which corresponds to the architecture in Fig. 11. The tool can generate pipelined and non-pipelined implementations of the same Megablock. The pipelined modules use the overlapping schedule described in Section III. The current version of the VHDL generation tool does not deal with Megablocks with memory operations. Note that non-pipelined Megablocks have been already tested and evaluated using an FPGA board (see [14]), and there is recent work focused on adding support to memory operations [16].

We start by considering a set of simple benchmarks without memory operations, named as *memoryless* (*compress1, count, even_ones, expand, fibonacci, hamming_dist, popcmpr, reverse* and *gcd1*). We implemented a single Megablock per benchmark. For this set we synthesized two versions of the coprocessor hardware module, with and without pipelining, to measure resource usage, confirm the execution clock cycles, and test the approach. A second set of 61 benchmarks named *embedded* include memory operations. The source codes of the benchmarks were examined to guarantee that inter-iteration memory dependencies were not present (future work will consider an automatic analysis). For this set we present overall speedup estimations for non-pipelined and pipelined Megablocks. For the overall speedup estimations, we consider a system architecture composed of a coprocessor coupled to the GPP. We use a MicroBlaze [9] as the target GPP with direct communication links to the coprocessor through FSL connections. The overall application speedups consider all communication overheads.

All benchmarks were compiled using *mb-gcc* 4.1.2 with the O2 flag. For the hardware modules we selected a Xilinx Spartan-6 LX45 FPGA as target. We consider a coprocessor with a maximum of 8 arithmetic/logic and 2 load/store units per row. Each of the arithmetic/logic units can be used to signal Megablock exits. We assume execution clock cycles of the operations equal to the ones requires by MicroBlaze equivalent instructions, when the processor is optimized for speed [9]. As with other approaches [1], we assume each memory operation can be completed in a single clock cycle. We also consider that the coprocessor is coupled to local memories allowing up to two simultaneous memory accesses per clock cycle (e.g., dual-port BRAMs). We use both the GPP and the coprocessor clocked at the same frequency.

Table I presents a comparison between the overall speedup obtained by the *memoryless* set, when considering non-pipelined and pipelined Megablock implementations. It contains information about the speedup, and the Critical Path Length (CPL) of the non-pipelined and pipelined designs. On average, pipelining Megablocks represents an increase in speedup of 1.4× (from 1.6× to 2.2×), over the non-pipelined implementation. However, in roughly half the cases of performance degradation occur after pipelining. With respect to the resource increase between FPGA designs (Fig. 12), in all cases, the pipelined implementations used more FFs (flip-flops), between 1.3× and 1.7×, and generally, the number of LUTs (look -up tables) also increased (between 1.01× and 1.8×).

Table I. Comparing non-pipelined with pipelined architectures using the memoryless set.

Benchmark	Non-Pipelined Speedup	Pipelined Speedup	Non-Pipelined CPL	Steady State Delay	Avg. Iter. per call
compress1	**1.65**	1.54	4	4	29
count	**1.72**	1.64	3	3	31
even_ones	1.68	**2.07**	3	2	31
expand	**1.66**	1.54	4	4	29
fibonacci	2.32	**6.83**	3	1	2378
hamming_dist	1.66	**2.04**	3	2	31
popcmpr	1.00	**1.16**	4	2	8.4
reverse	**1.88**	1.79	3	3	31
gcd1	0.97	**1.22**	8	6	166.2
average	1.62	**2.20**	3.89	3	303.9

The increase in resources and the performance degradation can be explained by the characteristics of the benchmarks. For the pipelining technique to be able to provide speedup over the non-pipelined versions, the latency of the steady state of the pipelined version must be lower than the CPL of the non-pipelined design. According to Table I, in all cases where the CPL remained the same, there was performance degradation. Having the CPL of the pipelined module very close to the CPL of the non-pipelined module indicates that the IM is replicating most of the critical path of the LM. As these are small benchmarks, the critical path represents a large portion of the Megablock body. We expect that in examples with memory accesses, the IM represents a much smaller portion of the Megablock body (e.g., mostly related to updates to the memory offsets), leading to lower increases in the resources needed for the pipelined designs. Fig. 13 presents overall application speedups for the set *embedded*. Without Megablock pipelining, we get slowdowns from 0.4× to speedups of 7.3× and an average speedup of 2.5× (or 2.1×, when using the geometric mean). With Megablock pipelining, we get slowdowns from 0.2× to speedups of 32×, with an average speedup of 5.6× (or 3×, when using the geometric mean). We observe that in several cases (16 in this set, 26% of the benchmarks) the pipelining contribution amplifies coprocessor speedups by a factor of 2 or more. E.g., we estimate a speedup of 3× for *vecsum* before pipelining and a speedup of 5.8× with an overlapping schedule.

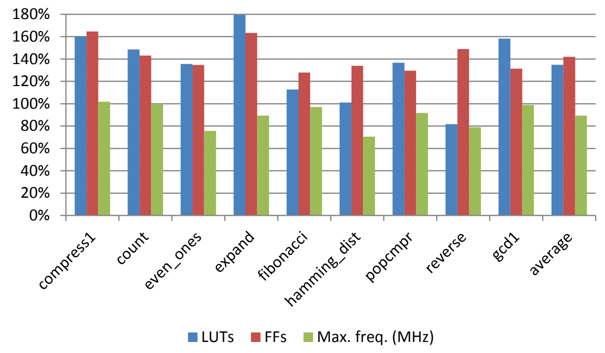

Figure 12. FPGA resources increase when using pipelining with overlapping schedule over the non-pipelined implementation.

When considering benchmarks with memory accesses, we achieve noticeable speedup improvements after pipelining (see columns *Speedup Improvement* in Table II). For instance, *change_brightness* and *compositing* went from a speedup of *1.6*× to a speedup of *9.3*×; *checkbits*, from *4.1*× to *12.4*×; *compress2*, from *2.5*× to *32*×. These improvements can be explained by two factors, presented in Table II. The first factor is the ratio between the average CPL of the executed Megablocks before pipelining (*CPL Non-Pipelining* column), and the average number of cycles of the steady state when Megablocks execute using the overlapping sche-

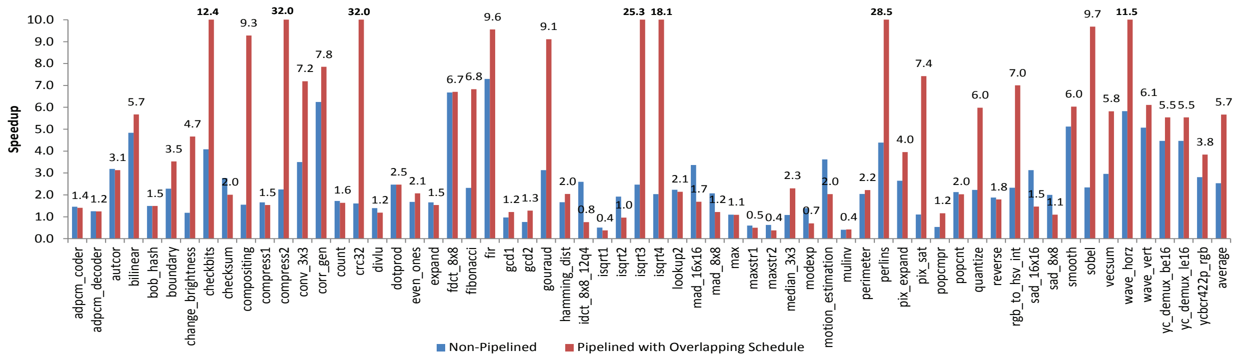

Figure 13. Individual overall speedups for a non-pipelined and a pipelined architecture with overlapping schedule, considering a maximum of 8 parallel arithmetic/logic FUs and 2 load/store operations per clock cycle.

dule (*Steady State Latency* column). The ratio between these two values (*CPL/Latency Ratio* column) is an upper-bound for the possible increase in speedup when applying pipelining. E.g., *crc32* went from a speedup of 1.6× to a speedup of 32× after pipelining, which represents an improvement of *20*×. The corresponding ratio is *24.5*×. The second factor is the number of average iterations per Megablock call (last column). Note that for all examples in Table II, the number of average iterations is high (above 99). When using pipelining, the improvement comes from the steady state execution. The higher the portion of execution is spent in the steady state (instead of the prologue), the closer the improvement is to the upper bound speedup given by the ratio between the baseline CPL and the steady state latency.

Table II. Sample CPL comparison between non-pipelined and pipelined with overlapping schedule.

Benchmark	CPL Non-Pipelined	Steady State Delay	CPL/ Latency Ratio	Speedup Improve-ment	Avg. Iter. p/ call
change_b.	12	2	6	5.81	99
checkbits	16	4	4	3.02	166
composit-ing	15	2	7.5	5.81	199
compress2	65	4	16.3	12.8	999
crc32	49	2	24.5	20	109
fibonacci	3	1	3	2.96	2,378
gouraud	6	2	3	2.94	1,999
isqrt3	112	2	56	10.12	99
isqrt4	73	2	36.5	9.05	99
pix_sat	14	2	7	6.73	2,000
quantize	6	2	3	2.69	199
rgb_to_hsv	55	16	3.4	3.04	499

VI. RELATED WORK

Dynamic partitioning and mapping is being increasingly focused by researchers. In the context of dynamic partitioning and mapping, loop pipelining, possibly due to its complexity, has not attracted the desired attention. To the best of our knowledge, our work is one of the first approaches considering loop pipelining in the context of dynamic partitioning and mapping. In this section we present relevant dynamic partitioning and mapping schemes in the context of reconfigurable computing systems. We also introduce some relevant loop pipelining approaches.

A. Approaches considering Dynamic Partitioning

We present here three approaches considering dynamic partitioning and mapping schemes in the context of reconfigurable computing systems. Specifically, we briefly describe the Warp Processor [2, 17], the DIM Reconfigurable System [1, 18], and the CCA [3, 19].

The Warp Processor [2, 17] is a runtime reconfigurable system which uses a custom fine-grained reconfigurable hardware (dubbed W-FPGA) as a hardware accelerator for a GPP. The system performs all steps at runtime, and attains significant speedups for benchmarks with bit-level operations. They report an average speedup of 6.3× over a set of 15 benchmarks.

The DIM Reconfigurable System [1, 18] proposes a reconfigurable array of FUs in a mesh-like topology and transparently maps single basic blocks (an improved version also considers speculation) from a MIPS processor to the array. They report an average speedup of 2.5× over a subset of the MiBench suite.

The CCA [3, 19] is composed of a reconfigurable array of FUs in an inverted triangular shape, coupled to an ARM processor. CCA work addresses control-data flow graph detection and mapping. They report an average speedup of 2.3× over a set of 29 benchmarks. Warp is the only approach of the three which considers entire loops, while CCA and DIM present speedups by only exploiting ILP using a small number of basic blocks. To the best of our knowledge, however, Warp does not consider loop pipelining.

B. Loop Pipelining

Loop pipelining, also known as software pipelining [12, 13], is a technique which has been extensively studied and proved to attain substantial increases in performance. The technique has been extensively applied in the context of both software and hardware compilers. It is a technique that must be part of every relevant high-level synthesis tool.

Loop pipelining is also important when mapping computations to CGRAs (see, e.g., [20]).

In the context of static binary compilation, Paek *et al.* [21] propose an approach which detects loops by performing static analysis of the binary. In their work they decompile the code and analyze loop structures. They focus on innermost loops, without branches and whose iteration count can be determined statically. They also consider loop unrolling when the iterations of both the inner and the outer loop can be determined statically (only the inner loop is unrolled). The detected loops are mapped offline to a data-flow oriented CGRA which supports context pipelining. After loops are detected, the binary is modified to include the CGRA mapping and communication routines. They report an average speedup of 9.4× when using examples of the DSPstone benchmark suite [22].

However, to the best of our knowledge, our approach is one of the first to address loop pipelining in the context of a dynamic partitioning system.

VII . CONCLUSION

We presented a technique for pipelining loops when transparently mapping computations from a GPP to a coprocessor unit. We took advantage of the characteristics of a particular runtime loop, the Megablock, to simplify the creation of pipelined loops (e.g., the Megablock loop contains only one control path). We further explored loop pipelining techniques under these circumstances, such as avoiding the implementation of an epilog by using atomic loop iterations, or delay memory store operations to the end of the iteration to avoid output dependencies, simplifying the implementation of atomic iterations.

We performed an analytical analysis on the conditions necessary for having increased performance with this technique, and suggested an architecture supporting this kind of pipelining. We evaluated two different approaches and sets of benchmarks: a first one which uses hardware implementations and simulation, and a second one relying on simulation and estimations. In a set of benchmarks without memory operations we increased the average speedup from 1.6× to 2.2×, after applying loop pipelining. In a set of 61 benchmarks with memory accesses the average speedup increased from 2.5× to 5.6×. For some particular benchmarks, the speedup improvements ranged from 3× to 20×, when compared with the non-pipelined version.

Dynamic partitioning can be a useful technique that enables taking advantage of reconfigurable hardware transparently. Although it is unlikely that an approach for automatic optimization of general computations will have better results than a handcrafted solution, the improvements achieved by dynamic partitioning can be good enough to justify the approach. Using loop pipelining is therefore an important technique to reach this goal. Future work will address the runtime analysis of Megablocks, using schemes to identify suitable Megablocks for pipelining and streaming based mapping techniques.

ACKNOWLEDGEMENTS

The authors acknowledge the partial support by the European Community's Framework Programme 7 (FP7) under contract No. 248976. We would like to acknowledge the fruitfull discussions with Nuno Paulino regarding his Megablock-based system prototypes.

REFERENCES

[1] A. C. Beck, M. B. Rutzig, G. Gaydadjiev, et al., "Run-Time Adaptable Architectures for Heterogeneous Behavior Embedded Systems," in Proc. 4th Intl. Works. Reconf. Comput.: Architectures, Tools and Applications, 2008, pp. 111-124.

[2] R. Lysecky, G. Stitt, and F. Vahid, "Warp Processors," ACM Trans. Des. Autom. Electron. Syst., vol. 11, pp. 659-681, 2006.

[3] N. Clark, J. Blome, M. Chu, et al., "An Architecture Framework for Transparent Instruction Set Customization in Embedded Processors," in Proc. 32nd Ann. Intl. Symp. Computer Architecture (ISCA'05), 2005, pp. 272-283.

[4] S. Hauck and A. DeHon, Reconfigurable computing: the theory and practice of FPGA-based computation: Morgan Kaufmann Pub, 2008.

[5] T. Wiangtong, P. Y. K. Cheung, and W. Luk, "Hardware/Software Codesign: a Systematic Approach Targeting Data-intensive Applications," IEEE Signal Processing Magazine, vol. 22, pp. 14-22, 2005.

[6] G. Stitt, F. Vahid, and S. Nematbakhsh, "Energy savings and speedups from partitioning critical software loops to hardware in embedded systems," ACM Transactions on Embedded Computing Systems (TECS), vol. 3, pp. 218-232, 2004.

[7] V. Baumgarte, G. Ehlers, F. May, et al., "PACT XPP—A Self -Reconfigurable Data Processing Architecture," J. Supercomput., vol. 26, pp. 167-184, 2003.

[8] J. Bispo and J. M. P. Cardoso, "On Identifying and Optimizing Instruction Sequences for Dynamic Compilation," in Proc. Intl. Conf. on Field-Programmable Tech., Beijing, China, 2010, pp. 437-440.

[9] I. Xilinx, "Microblaze Processor Reference Guide v13.4," reference manual, 2011.

[10] J. Hennessy, D. Patterson, D. Goldberg, et al., Computer architecture: a quantitative approach: Morgan Kaufmann, 2003.

[11] E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, et al., "Chapter 9 Sequencing and scheduling: Algorithms and complexity," in Handbooks in Operations Research and Management Science. vol. Volume 4, A. H. G. R. K. S.C Graves and P. H. Zipkin, Eds., ed: Elsevier, 1993, pp. 445-522.

[12] B. R. Rau, "Iterative modulo scheduling: An algorithm for software pipelining loops," 1994, pp. 63-74.

[13] V. Allan, R. Jones, R. Lee, et al., "Software pipelining," ACM Computing Surveys (CSUR), vol. 27, pp. 367-432, 1995.

[14] J. Bispo, "Mapping Runtime-Detected Loops from Microprocessors to Reconfigurable Processing Units," PhD, Instituto Superior Técnico, 2012.

[15] G. Kuzmanov, G. Gaydadjiev, and S. Vassiliadis, "The MOLEN processor prototype," in 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), 2004, pp. 296-299.

[16] N. Paulino, J. C. Ferreira, and J. M. P. Cardoso, "Architecture for Transparent Binary Acceleration of Loops with Memory Accesses," presented at the 9th International Symposium on Applied Reconfigurable Computing (ARC'2013), Los Angeles, USA, 2013.

[17] R. Lysecky and F. Vahid, "Design and implementation of a MicroBlaze-based warp processor," ACM Transactions on Embedded Computing Systems, vol. 8, pp. 1-22, 2009.

[18] A. C. S. Beck, M. B. Rutzig, G. Gaydadjiev, et al., "Transparent reconfigurable acceleration for heterogeneous embedded applications," in Proc. Conf. Design, Automation and Test in Europe (DATE'08), Munich, Germany, 2008, pp. 1208-1213.

[19] N. Clark, M. Kudlur, H. Park, et al., "Application-Specific Processing on a General-Purpose Core via Transparent Instruction Set Customization," in Proc. 37th Ann. IEEE/ACM Intl. Symp. Microarch., Portland, USA, 2004, pp. 30-40.

[20] B. Mei, S. Vernalde, D. Verkest, et al., "ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix," in Field-Programmable Logic and Applications, ed, 2003, pp. 61-70.

[21] J. K. Paek, K. Choi, and J. Lee, "Binary acceleration using coarse-grained reconfigurable architecture," ACM SIGARCH Computer Architecture News, vol. 38, pp. 33-39, 2011.

[22] V. Zivojnovic, J. M. Velarde, C. Schlager, et al., "DSPstone: A DSP-oriented benchmarking methodology," in Proc. of the Intern. Conf. on Signal Processing and Technology, 1994, pp. 715-720.