This supplementary file elaborates on the MLCA programming model that motivates our design of the CP microarchitecture. It also gives additional details about the resource allocator described in Section 4.6 of the main paper. Finally, it indicates how the Verilog code of our CP design may be downloaded.

1 THE MLCA PROGRAMMING MODEL

An overview of the MLCA architecture and its programming model is given by Karim et al. [1]. The goal of this section is to elaborate on the features and the flexibility of the programming model. It is these features that motivate our design of the CP microarchitecture. Section 1.1 provides background on the programming model. Section 1.2 gives more details on breaking of false dependences in the context of the example shown in Figure 2 of the main paper. Section 1.3 explains how MLCA programs can embody dynamic task dependences.

1.1 Background

An MLCA program consists of a control program and set of task functions.

The control program is written in a high-level language called Sarek. An example control program is given in Figure 1. The structure of a Sarek closely resembles the structure of a C program. Sarek features task calls, flow-control statements (e.g., while, if) and two types of registers. The data registers (reg_t) are used to pass data between tasks and control registers (cr_t) are used for control flow conditions. Task calls are similar to C function calls. A task call specifies the name of the task to be called, the registers it reads and writes (designated with in and out), and optionally a control register it writes. For example the QAM task in Figure 1 call reads var1 and writes to var2 and var3 data registers.

```
task Config, CR1
TOP:
task GetFrame, CR2, R1:w
if (CR2 == false) jmpa END
task QAM, R1:r, R2:w, R3:w
task CIC_I, R2:r, R4:w
task CIC_Q, R3:r, R5:w
task Demod, R4:r, R5:r, R6:w
if (CR1 == false) jmpa IF_FALSE
task Filter, R6:r, R7:w
jmpa IF_END
IF_FALSE:
task Copy, R7:r, R7:w
IF_END:
task Decimate, R7:r, R8:w
task Output, R8:r
jmpa TOP
END:
stop
```

Fig. 2. Assembly code of the program from Figure 1
void Demod() { 
    int var1, var2, var3;
    var1 = readArg(1);
    var2 = readArg(2);
    // do some work
    var3 = ...
    writeArg(1, var3);
} 

void GetFrame() { 
    int var, cond;
    // do some work
    var = ...
    cond = ...
    writeArg(1, var);
    if (cond) 
        writeArgC(1);
    else 
        writeArgC(0);
} 

Fig. 3. Example MLCA task functions for the control program in Figure 1 

The GetFrame task writes to var1 data register and dataAvail control register.

The assembly code of the control program is shown in Figure 2. It contains task instructions (task) that represent task calls and CP internal instructions that implement control flow (conditional and unconditional jumps, jmpa). The data and control registers from Sarek are mapped to URF (R1 to R8) and CRF (CR1, CR2) registers respectively.

Figure 3 outlines the task functions that correspond to GetFrame and Demod tasks. Task functions can be implemented in any high-level language that can be compiled for the PUs, in this case C. The task functions do not have any formal parameters. They access task URF and CRF arguments through a special API that contains three functions:
- int readArg(int arg_num),
- void writeArg(int arg_num, int val),and
- void writeArgC(int val).

For example, the readArg(1) and readArg(2) function calls in Demod will read the task’s first and second argument which correspond to R4 and R5 URF registers, respectively. The writeArg(1, var3) function call writes the value of var3 to task’s first argument which corresponds to R6. The writeArgC(1) in GetFrame writes 1 to task’s control argument which corresponds to CR2 register.

The MLCA programming model allows expression of programs in a natural sequential-like fashion. The control program is a sequential program of task calls. Similarly, task functions are sequential C functions extended to read and write their arguments. The constructs of the model facilitate the expression of realistic applications as MLCA programs, either from scratch or by porting existing sequential C applications. There are no restrictions on the functionality that a task function can implement. Thus, task functions can have variable number of inputs and outputs. Further, the reading and writing of arguments within a task function can be made at arbitrary locations and in any order, possibly interleaved with one another. Writing an output as soon as it is produced may allow waiting tasks to execute earlier, revealing more parallelism. This is illustrated with the example in Figure 4, which shows a simplified task function called Preprocess which has 19 inputs and 11 outputs. For clarity, only some inputs and outputs are shown within the task function. The task function reads some input arguments, carries out some computation and writes some output arguments, and then repeats the process. In addition, the order of reads and writes may also depend on control flow within a task function, in this example it depends on the run-time value of the cond variable. It is this flexible nature of the MLCA programming model that gives rise to the constraints on the design of the CP microarchitecture described in Section 3 of the main paper.

1.2 Breaking false dependences

In this section we elaborate on the process of breaking false dependences and discovering parallelism in the context of the example shown in Figure 2 of the main paper. The example contains false dependences among different iterations of the loop. Within the loop body itself, there are no false dependences. The left graph of Figure 2 shows false dependences caused by $A_2$. These are: WAW with $A_1$ because of $a$, WAR with $B_1$ because
of $a$, and WAR with $C_1$ because of $a$. $B_2$ generates the following false dependences: WAR and WAW with $B_1$ because of $b$, and WAW because of $b$, WAR with $C_1$ and $D_1$ because of $b$. Similar false dependences exist for $C_2$, $D_2$, and $E_2$.

Figure 5 shows parallelism that can be exploited across multiple iterations, after the CP has removed all the above false dependences. After 4 iterations have been dispatched into the CP pipeline the execution will reach steady state in which there is always 5 tasks that can be executed in parallel (in this example we assume all tasks have equal latency). In the first round of the steady-state, $E_1$, $C_2$, $D_2$, $B_3$, $A_1$ execute, in the next round $E_2$, $C_3$, $D_3$, $B_4$, $A_2$ execute and so on.

1.3 Dynamic Dependences and Scheduling

The example in the previous section gives rise to false dependences that are regular across the iteration space and can arguably be determined and handled by static analysis using a compiler. However, the MLCA programming model can give rise to dependences that have a more dynamic nature and that can only be conservatively estimated by static analysis, leading to loss of parallelism. In these cases, the use of the out-of-order execution CP exploits more parallelism than static analysis. In this section, we provide one example of such programs.

Consider the MLCA control program shown in Figure 6. It consists of a loop with two if-the-else statements in it body. The program executes either T1 or T2 based on cond1 followed by either T3 or T4 based on cond2. T3 reads the output written by T1 while T4 reads the output written by T2. Similarly, T1 reads the output written by T3 while T2 reads the output written by T4. However, the dependences that arise during execution depend on the two run-time conditions, cond1 and cond2.

Figure 7 shows the (true) dependences that arise for two scenarios of values of cond1 and cond2 across three iterations of the while loop. In the figure, the tasks on the same row can execute in parallel and values of cond1 and cond2 are given for each iteration. For example, in Figure 7a cond1 and cond2 are respectively false (F) and false (F) in the first and second, and T and F in third iteration. This serializes execution of T2 and T4 across iterations. T3 from the third iteration can execute early, since it depends on T0 which is the most recent task that wrote b. In contrast, in Figure 7b cond1 and cond2 evaluate to true (T) and true (T) in the first, F and T in second, and T and F in third iteration. This results in less serialization and consequently in more parallelism. It is clear that the instances of the tasks that may execute in parallel and the serialization among tasks instances are a function of the exact values of cond1 and cond2 at run-time.

Pure static analysis must be conservative and thus must assume dependences across the iterations of the loop because of the usage of variables e and f by the tasks. Thus, execution based on static analysis would serialize the tasks across the iterations, whether such cross-iteration dependences exist during execution (Figure 7a) or not (Figure 7b) leading to loss of parallelism in the latter case. In contrast, the run-time detection and resolution of dependences uncovers the true dependences that exist among the tasks based on which tasks actually execute, leading to more parallelism. Furthermore, it is unclear that static analysis can be successful at register renaming and breaking false dependences across loop iterations, particularly in the presence of dependences between tasks that are several iterations apart. While it may be possible to extend static analysis with run-time support, such support evolves to become a software implementation of out-of-order execution in order to han-
Fig. 7. Data-flow graphs showing two scenarios of dynamic dependences across 3 iterations of loop from Figure 6

dle all possible scenarios of renaming and dependences. Capalija [2] shows that such software implementation can become a bottleneck.

The ability of the MLCA to capture dependences dynamically greatly increases the flexibility of the programming model, making it easier to express realistic applications. However, such flexibility precludes the use of static analysis and necessitates the use of dynamic out-of-order techniques to uncover more parallelism. This is not unlike the need for such techniques that uncover dynamic parallelism among instructions in superscalar processors.

Furthermore, the CP schedules tasks dynamically. This is beneficial in the presence of run-time latency variation of tasks. For example, task’s latency might vary across iterations due to the control flow internal to the task. The CP schedules tasks to available PUs as soon as they are ready, hence it will seamlessly adapt to any latency variation. Dynamic scheduling is especially important for tasks on the critical path (which itself could be dynamic), those tasks have to be scheduled back-to-

back as soon as they become available, which is hard to predict at compile-time.

2 RESOURCE ALLOCATOR DETAILS

We elaborate on the reasons why physical registers are indeed released out-of-order. The physical registers are released at task retire, which is itself in-order, however the release criteria causes physical registers to be released out-of-order. We illustrate this using an example. A physical register $P_5$ assigned to an output architectural register $R_1$ of task $T_A$ is not released during $T_A$ retire, rather upon retire of the next task in program order (e.g., $T_B$) that writes to same architectural register $R_1$. In fact, the time period between allocation of $P_5$ and its release can be arbitrarily long. In the extreme case $T_A$ can be the first task in the application, and $T_B$ can be the last one. Between the execution of the $T_A$ and $T_B$, other tasks can write to other registers (e.g. $R_3$, $R_4$), so physical registers assigned to those architectural registers will be released before $P_5$, resulting in an out-of-order release.

Figure 8 shows the resource usage of the three alternative allocator implementations: bit-vector, bit-array, and FIFO-based allocator. The design of the bit-vector and bit-array allocators are described in detail in Section 4.6 of the main paper. The FIFO-based allocator uses an $N$ entries deep and $\log_2(N)$ bits wide FIFO to store free physical register IDs (or RDLT entry IDs). The resource usage results were obtained on our FPGA platform described in Section 5.1 of the main paper. The bit-vector allocator consumes a large amount of logic resources, proportional to the size of the allocator ($O(N)$). On the other hand, the FIFO-based approach consumes exactly $O(N \times \log_2(N))$ memory bits. Our bit-array allocator implementation uses a balance of both types of resources, but modest amount of each compared to other two alternatives. The array-based logic resource count grows proportionally to $O(\sqrt{N})$, and memory bits are proportional to $O(N)$. 

Fig. 8. Memory and logic resource usage of three alternative resource allocator implementations
3 Code Release

The CP Verilog code we developed is released at http://www.eecg.toronto.edu/~davor/MLCA. We hope that it will serve both as a platform for those wishing to pursue similar research on coarse-grain architectures and as an FPGA circuit benchmark.

References