COMPILER SUPPORT FOR A MULTIMEDIA SYSTEM-ON-CHIP ARCHITECTURE

by

Utku Aydonat

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto

Abstract

COMPILER SUPPORT FOR A MULTIMEDIA SYSTEM-ON-CHIP ARCHITECTURE

Utku Aydonat
Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto
2005

The *Multi-Level Computing Architecture (MLCA)* is a novel parallel System-on-Chip architecture targeted for multimedia applications. Although it provides a simple programming model that eases porting of applications, the architecture requires the support of a compiler to deliver good performance. We design code transformations that increase the performance of MLCA programs. These code transformations are parameter deaggregation, buffer privatization, buffer replication and buffer renaming. We implement the code transformations in a prototype compiler which is based on the ORC compiler. We also provide an API for programmers to optionally give high level data access information to the compiler. Our experimental evaluation of the prototype compiler, using an MLCA simulator and real multimedia applications, shows that our code transformations generate MLCA programs that exhibit scaling speedups comparable to that of the manually ported versions of the applications.

Acknowledgements

I would like to thank Prof. Abdelrahman for his great support thoughout the design, implementation and documentation of this work. His vision enabled me to continue on the right path from the beginning, and with his feedback I was able to solve many critical problems.

I would also thank Faraydon Karim and Alain Mellan for introducing me to the MLCA and for their unweavery support during my work, in particular with the MLCA simulator and applications. They have spent many hours over the phone explaining and discussing the MLCA architecture and the simulator. Faraydon Karim provided valuable feedback about architectural issues related to the MLCA. Alain Mellan created and maintained the MLCA simulator, and quickly responded to my questions about the MLCA programming model and the simulator.

This work has been supported in part by research grants from STMicroelectronics, and by Communications and Information Technology Ontario (CITO). I am very grateful for their support.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

A *System-on-a-Chip (SoC)* is an integrated design that incorporates programmable cores, custom or semi custom blocks, and memories into a single chip [9]. This architecture allows reuse of pre-designed cores (commonly referred to as intellectual property, or IP), thus amortizing the design cost of a core over many system generations. It is no surprise then that SoCs have become a solution paradigm for processing in many consumer products, such as mobile phones, laptops and PDAs [25].

The increasing complexity of today's embedded applications, particularly multimedia and network routing, has been exerting considerable demands on embedded processors. Consequently, SoC designers have resorted to the use of multiple programmable processing elements in a single SoC. Examples of such *parallel* SoCs include systems from Daytona [13], picoChip [8] and Cradle Technologies [1].

However, these parallel SoCs pose programming challenges to the application developers. This is mainly because programming parallel systems is considerably more complex and time consuming than programming single-processor systems. This increased effort in programming results in increased software costs and lower productivity.

The *Multi-Level Computing Architecture (MLCA)* is a novel SoC architecture currently being developed by STMicroelectronics [21]. It is intended to provide a simple programming model while delivering high performance for multimedia applications. It features multiple processing units, a shared memory and a top level controller that uses well-developed superscalar principles to automatically exploit parallelism among coarse-grain units of computation, or *tasks*. The MLCA supports a programming model, which, similar to that of sequential programming, does not require programmers to specify synchronization and/or data communication. In this model, an application consists of tasks (written in regular high-level languages such as C) and a high-level *control program* that represents the control and data flow among these tasks. The usage of data among tasks is explicit in the form of the input and output arguments of the tasks. Subsequently, the hardware is able to rename the input and output arguments of tasks and schedule in parallel the tasks that do not have data dependences with each other.

In spite of its features, the MLCA requires the support of a compiler to alleviate performance problems that may arise in multimedia applications. These problems are mainly caused by the usage of pointers to aggregate data in shared memory, which limits the ability of the MLCA hardware to exploit parallelism. The focus of this work is such compiler support.

More specifically, this thesis describes the MLCA Optimizing Compiler, intended to assist the programmers in porting applications to the MLCA. The target class of applications for the MLCA Optimizing Compiler is multimedia applications. These applications have simple control flow, usually a main loop consisting of calls to work functions and which iterates until the end of input. Furthermore, multimedia applications generally use simple data structures, mostly buffers (arrays) and structures. The data flow between the work functions is realized by passing pointers to these buffers and structures.

We design several code transformations for the MLCA Optimizing Compiler, which benefit from the common properties of the multimedia applications. These code transformations are referred as parameter deaggregation, buffer privatization, buffer replication and buffer renaming. They enable parallel execution of tasks as well as handle correctness issues in control programs. They are based on standard compiler analyses such as inter-procedural data-flow and array section analyses. Furthermore, it is possible for programmers to provide high-level data usage information, in the form of pragmas, to complement these compiler analyses. We believe that our code transformations are applicable and beneficial to most multimedia applications, and they greatly reduce the programmer's effort in porting applications to the MLCA. On the other hand, the characteristics of these applications may limit the applicability of the code transformations. Parameter deaggregation can not be applied to recursive structures, i.e. the structures that contain pointers to each other, such as linked lists. In addition, buffer privatization, buffer replication and buffer renaming can not be applied or can only be applied conservatively, in case the control program contains loops that access different sections of buffers in different iterations.

We implement a prototype of the MLCA Optimizing Compiler to evaluate our code transformations. We use Open Research Compiler (ORC) [7] as the infrastructure for analyzing the tasks and applying the code transformations to control programs and tasks. We also implement an API that allows programmers to insert/modify the prerequisite analyses results. This feature of our compiler is useful when ORC analyses fail or are too conservative because of compile time restrictions (such as I/O operations, aliasing, etc.).

We experiment with three multimedia programs that represent the class of applications that the MLCA Optimizing Compiler is targeted for. The results show that applying the code transformations results in application performance that is comparable to that of hand-ported applications.

1.1 Thesis Contribution

This thesis makes the following contributions:

1. *Code Transformations:* This thesis introduces code transformations for creating correct and high performance MLCA programs. The legality and the effectiveness of these code transformations are discussed and their implementation is described in detail.

2. *Design and Implementation of a Prototype Compiler:* This thesis describes the design and implementation of an optimizing compiler targeting the MLCA. The phases and the user API of this compiler are designed and implemented with an open-source compiler infrastructure.

3. *Experimental Evaluation of the Code Transformations:* This thesis provides an experimental evaluation of the code transformations. The overall and the individual benefits of these code transformations are shown and their overheads are assessed.

In addition, this thesis provides practical knowledge about porting applications to the MLCA, by stating possible performance bottlenecks and critical programmer mistakes in MLCA applications. A technique for porting applications to the MLCA programming model is also suggested in this thesis. Furthermore, by presenting experimental results for three multimedia applications, this thesis gives an indication of the performance of the MLCA.

1.2 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 gives a brief a background on the concepts used in the thesis. Chapter 3 describes the target architecture, i.e. the MLCA. Chapter 4 discusses the need for compiler support for the MLCA. Chapter 5 introduces the code transformations designed for the MLCA Optimizing Compiler. Chapter 6 details the algorithms of the code transformations. Chapter 7 presents the overall structure of the MLCA Optimizing Compiler. Chapter 8 presents and discusses the experimental results for the MLCA Optimizing Compiler. Chapter 9 discusses the related work. Chapter 10 concludes and gives directions for future work. Appendix A presents some of the implementation issues for the MLCA Optimizing Compiler.

Chapter 2

Background

In this chapter, we review some of the basic concepts related to superscalar processors, data dependence, control dependence, out-of-order execution, speculative execution, register renaming, aliasing, array section analysis and array privatization.

2.1 Superscalar Processors

Superscalar processors are a genre of multiple-issue processors, in which varying number of instructions are issued per clock cycle. The eligibility of instructions for issue is checked by the fetch unit of the superscalar processor, in the program order. Eligible instructions are issued to the computation unit, allowing the execution of multiple instructions in a cycle, effectively decreasing the cycle-per-instruction (CPI) to below one and exploiting more instruction-level parallelism.

Most of the today's processors are based on superscalar techniques. Sun UltraSPARC II/III, IBM Power2, Pentium III/4 and MIPS R10K are among the most well-known commercial superscalar processors.

2.2 Data Dependences

Data Dependence is said to exist between two instructions if they access the same data (in a register or in memory) and at least one of them writes this data. Data dependence is classified into three categories; *flow*, *output* and *anti* dependences [17].

Flow dependence occurs when data is read by an instruction *after* it is written to by another. Flow dependences are a part of natural program data-flow and in general can not be eliminated. For this reason, flow dependences are also called *true dependences.* A flow dependence between two instructions $\mathtt{i}_1$ and $\mathtt{i}_2$ is identified by $\mathtt{i}_1\delta^f\mathtt{i}_2$.

Output dependence exists when two instructions are writing to the same register or the same memory location. In order to preserve correctness, instructions have to write to the memory or register in the execution order, causing the second instruction to be issued only after the first one is completed. In other words, the execution of the instructions must be in program order, although there is no true dependence between them. Unlike true dependences, output dependences can be eliminated with renaming. Therefore, they are also referred to as *false dependences.* An output dependence between two instructions $\mathtt{i}_1$ and $\mathtt{i}_2$ is identified by $\mathtt{i}_1\delta^o\mathtt{i}_2$.

Anti dependence exists when a register or memory location is written to by an instruction, after it is read by another. To ensure correct execution, the second instruction must wait for the first one to read the data before it can write the data. Therefore, the instructions have to access the memory location or register in the program order, serializing their execution. Anti dependences are also false dependences in that they can be resolved using renaming. An anti dependence between two instructions $\mathtt{i}_1$ and $\mathtt{i}_2$ is identified by $\mathtt{i}_1\delta^a\mathtt{i}_2$.

Figure 2.1 depicts some examples of data dependences. In the figure, instructions are represented by an opcode followed by a destination register and two source registers. In Figure 2.1(a), there is a flow dependence $\mathtt{i}_1\delta^f\mathtt{i}_2$ between the two instructions, through register `R0`. In other words, the destination of the first instruction, i.e. `DIV`, is `R0` register,

```
(1)    DIV    R0, R2, R4
(2)    ADD    R10, R0, R8
```

(a) Flow dependence.

```
(1)    DIV    R4, R2, R0
(2)    ADD    R4, R0, R8
```

(b) Output dependence.

```
(1)    DIV    R2, R1, R4
(2)    ADD    R1, R0, R8
```

(c) Anti dependence.

Figure 2.1: Data dependence types.

which is also a source register for the second instruction, i.e. `ADD`. In Figure 2.1(b), the second instruction has to wait the first one, because they both write to register `R4`. This creates an output dependence $\mathtt{i}_1\delta^o\mathtt{i}_2$ between the two instructions. Figure 2.1(c) shows an example in which the two instructions have to be issued in-order because of the anti-dependence $\mathtt{i}_1\delta^a\mathtt{i}_2$ caused by the accesses to the register `R1`.

2.3 Control Dependences

Control Dependence is said to exist between two instructions, if the outcome of one instruction, determines whether the other instruction will be executed or not [17]. For instance, control dependences caused by the branch instructions determine the execution order of instructions. In that sense, every instruction, except for those in the first basic block of the program, is control dependent on some set of branches, and, in general these control dependences must be preserved for correct execution. If an instruction $\mathtt{i}_1$ is control dependent to another instruction $\mathtt{i}_2$, this is denoted by $\mathtt{i}_1\delta^c\mathtt{i}_2$.

```
if b1 {
      S1;
};
if b2 {
      S2;
};
```

Figure 2.2: Control dependences.

For example, in the code segment of Figure 2.2, instruction `S1` is control dependent on `b1` branch instruction, and `S2` is control dependent on `b2` but not on `b1`. In other words, there exist two control dependences `S1`δ^c`b1` and `S2`δ^c`b2`.

2.4 Out-of-Order Execution

Out-of-order execution is a technique used in superscalar processors, enabling the issue of instructions out of program order [17]. With out-of-order execution, instructions are checked for issue in program order and an instruction may be issued to computation units regardless whether all the instructions previous to it have been issued or not. Thus, in contrast to in-order execution, a stalled instruction does not cause its subsequent instructions to be stalled.

Figure 2.3 depicts a candidate code for out-of-order execution. In the figure, the first and the third instructions are division operations that take significantly longer than the second addition operation. While the first instruction is being executed, the second instruction can not be issued to the computation unit because of the true data dependence caused by the accesses to `R0` register. With in-order execution, the third operation, although it does not have any data dependence with either of the two previous instructions, is stalled until the first instruction is completed and second instruction is issued. The impact of this stall can be significant on the execution time, because the first division instruction can take many cycles to complete. In contrast, with out-of-order execution, the

```
(1)     DIV     R0, R2, R4
(2)     ADD     R10, R0, R8
(3)     DIV     R12, R8, R14
```

Figure 2.3: An example code for out-of-order execution.

third instruction will be issued without waiting for the completion of the first or second instructions and therefore overlapping the two `DIV` instructions, considerably decreasing the execution time (assuming that hardware support exists to allow overlapped execution of the two `DIV` instructions).

2.5 Speculative Execution

Speculative execution eliminates the effect of stalls caused by the control dependences [17]. This is achieved by allowing instructions to execute before the control dependences are resolved. If the speculation is not correct, the effects of the incorrectly speculated instructions are undone. However, if the speculation is correct, the results of the speculative instructions are committed. Therefore, speculative execution keeps a processor busy, which pays off when the speculation is correct, as well as opens up more opportunities for better dynamically scheduling instructions [17].

Figure 2.4 illustrates speculation using an example code sequence. In the example, the fifth instruction is a branch instruction and every instruction inside the loop is control dependent on this branch instruction. Strictly, it is not possible to know if the branch will be taken or not until the branch instruction is actually executed. Without speculation, instructions will be stalled until the branch is completed at the end of every loop iteration. However, with speculative execution, instructions are issued speculatively, even if the branch condition is not resolved, based on some predictions about the outcome of the branch. If the speculation is correct, improvement in the execution time will result.

```
(1)     LOOP:   LOAD    R0, 0(R1)
(2)             MUL     R4, R0, R2
(3)             STORE   R4, 0(R1)
(4)             ADDI    R1, R1, #-8
(5)             BNE     R1, R2, LOOP
```

Figure 2.4: An example code for speculative execution.

2.6 Register Renaming

Register Renaming is a technique used to resolve false dependences, i.e. output and anti dependences, caused by register accesses [17]. It eliminates these dependences by renaming all destination registers, including those with a pending read or write for an earlier instruction, so that the out-of-order write does not affect any instructions that depend on an earlier value of the operand.

Figure 2.5 gives an example of how renaming breaks output dependences. In Figure 2.5(a), there is an output dependence between the second and third instructions because they both write to register `R6`. The hardware or the compiler could replace the destination register of the latter instruction, i.e. `SUB`, with a new register, eliminating the output dependence. The Figure 2.5(b) shows the result of the renaming, when register `R6` is replaced with register `R10` in the destination of the `SUB`. Subsequently, register `R10` is also used in all the subsequent instructions instead of `R6`.

Figure 2.6 depicts renaming for anti-dependences. The anti-dependence in Figure 2.6(a) between `ADD` and `SUB` through register `R3` is resolved by renaming the destination of `SUB` with register `R10`, as in Figure 2.6(b). Similarly, `R10` is used in every subsequent instruction, instead of `R3`, to preserve the correctness. After the renaming, the hardware or the compiler can schedule the `SUB` instruction earlier than `ADD`, without violating program correctness.

```
(1)     DIV     R0, R2, R4
(2)     ADD     R6, R0, R8
(3)     SUB     R6, R2, R4
(4)     MUL     R7, R6, R4
```

(a) Before renaming.

```
(1)     DIV     R0, R2, R4
(2)     ADD     R6, R0, R8
(3)     SUB     R10, R2, R4
(4)     MUL     R7, R10, R4
```

(b) After renaming.

Figure 2.5: Renaming for output dependences.

```
(1)     DIV     R0, R2, R4
(2)     ADD     R6, R3, R8
(3)     SUB     R3, R2, R4
(4)     MUL     R7, R3, R4
```

(a) Before renaming.

```
(1)     DIV     R0, R2, R4
(2)     ADD     R6, R3, R8
(3)     SUB     R10, R2, R4
(4)     MUL     R7, R10, R4
```

(b) After renaming.

Figure 2.6: Renaming for anti dependences.

```
void Read_Data(FILE* file, int buf[])
{
  int i;

  for(i = 30; i <= 40; i++)
    buf[i] = fgetc(file);
}
```

Figure 2.7: An example code for array section analysis.

2.7 Alias Analysis

Two variables are said to be *aliases*, if they point to the same memory location. For instance, in the C programming language, if $p = \&i$, then it is said that $< *p, i >$ is an alias pair, meaning that the two elements reference data in the same memory location. The process of finding variable alias pairs in a program is called *alias analysis*. This analysis is very important in compilers, because it allows building more accurate data dependence relationships among the variables, allowing more aggressive optimizations and instruction scheduling. Alias analysis is a complex process, for which several approaches have been proposed [11, 18, 19].

2.8 Array Section Analysis

An *array section* describes the set of array elements that are either read or written by a group of program statements. Several approaches have been proposed to obtain array sections, differing in their efficiency and accuracy [16]. Generally, these approaches are referred as *array section analyses* and resulting array sections are represented with a pair of start offset and end offset inside brackets together with the type of the access. For instance, in Figure 2.7, `[30:40]:Write` is the section of the array `buf` accessed by the code fragment shown.

```
S1:     for(I = 1; I < N; I++) {
S2:        A[1] = X[I][J]
S3:        for(J = 2; J < N; J++) {
S4:           A[j] = A[J - 1] + Y[J];
S5:        }
S6:        for( K = 1; k < N; k++) {
S7:           B[I][K] = B[I][K] + A[K];
S8:        }
S9:     }
```

Figure 2.8: An example code for array privatization [27].

2.9 Array Privatization

Privatization [12] is a technique that allows concurrent threads or processors to allocate a scalar or an array in their respective private memory space, allowing different threads or processors to safely access this scalar or array without the need for synchronization. In other words, privatization eliminates memory related data dependence by providing a distinct instance of a scalar or an array to each thread or processor. If privatization is applied to a scalar variable, the transformation is called *scalar privatization*, whereas for arrays, it is called *array privatization*. In order for array privatization to be legal for an array A in a loop L, every fetch to an element of A in L must be preceded by a store to the element in the same iteration of L [27].

Figure 2.8 shows an example of array privatization. In the depicted example, each iteration of loop S1 accesses the same elements of array A, forcing the iterations of S1 to serialize. However, data written to A in one iteration of S1 is only accessed in the same iteration of S1. Therefore, marking A private to each iteration of loop S1 allows the parallel execution of each loop iteration.

A number of techniques have been proposed to privatize arrays including the one by Tu [27].

Chapter 3

The MLCA

The *Multi-level Computing Architecture (MLCA)* is a template architecture for SoC systems intended for multimedia and streaming applications. The MLCA features multiple processing units and a top level controller that automatically exploits parallelism among coarse-grain units of computation, or tasks, using well-developed superscalar principles.

In this chapter, we give an overview of the MLCA. Section 3.1 describes the architecture itself. Section 3.2 describes the programming model supported by the MLCA. Section 3.3 describes the renaming and synchronization mechanism of the MLCA. Section 3.4 argues the benefits of the MLCA and its programming model. Section 3.5 presents example of the performance of the MLCA (using a simulator) for two multimedia applications.

3.1 The MLCA

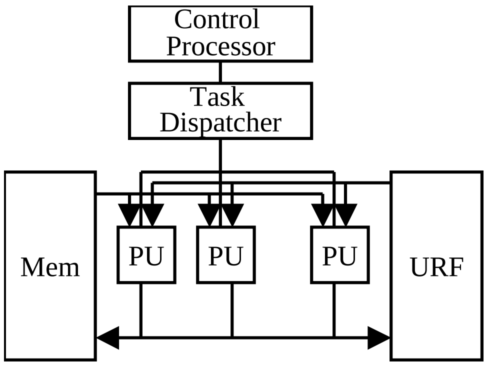
The MLCA is a 2-level hierarchical architecture that at the lower level consists of multiple processing units (PUs). A PU can be a full-fledged processor core (superscalar, VLIW, etc), a DSP, a block of FPGA, or some custom hardware. The upper level consists of a control processor (CP), a task dispatcher (TD), and a universal register file (URF). A dedicated interconnect links the PUs to the URF and to memory. A block diagram of the MLCA is shown in Figure 3.1(a). It bears considerable similarity to an abstract microprocessor architecture, in Figure 3.1(b).

The novelty of the MLCA architecture stems from the fact that the upper level of the hierarchy supports out-of-order, speculative and superscalar execution of coarse-grain units of computation, which is referred to as tasks. It does so using the same techniques used in today's superscalar processors, such as register renaming and out-of-order execution, to exploit parallelism among instructions. This leverages existing superscalar technology to exploit task-level parallelism across PUs in addition to possible instruction-level parallelism within a PU.

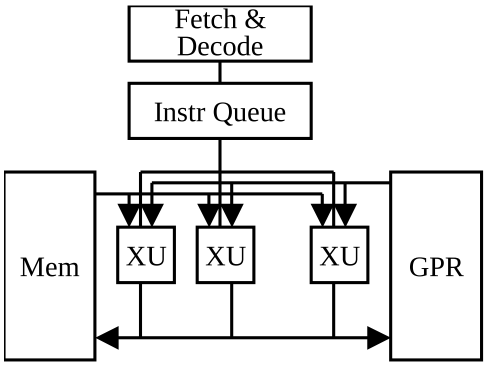
The CP fetches and decodes task instructions, each of which specifies a task to execute. A task instruction also specifies the inputs and outputs of the task as registers in the URF. Dependences among task instructions are detected in the same way that dependences among instructions are detected in a superscalar processor: by means of the source and sink registers in the URF. The CP renames URF registers as necessary to break false dependences among task instructions. Decoded task instructions are then issued to the TD unit. Based on dynamic dependences, tasks can be issued out-of-order, and may also complete and commit their outputs out-of-order.

Task instructions are enqueued in the TD unit, in a similar way instructions are enqueued in the instruction queue of a superscalar PU. When the operands of a task instruction are ready, the task instruction is dispatched using a scheduling strategy to the PUs. The simplest of such strategies dispatches instructions to PUs in a round-robin fashion. However, more dynamic strategies can also be used.

The MLCA is a template of an architecture. It does not specify the form of the interconnect among the PUs. Several implementations are possible, including buses, cross-bars, and multi-stage interconnects. In addition, since inter-task dependences are enforced by the CP, and since data communication is primarily accomplished through the URF, there is no need to assume a particular memory architecture. The PUs may



(a) Macro-architecture.



(b) Micro-architecture.

Figure 3.1: The MLCA high-level architecture analogy.

share a single memory, or may each have its own private memory, or any combination of the two, depending on the application. However, a shared memory accessible by all the PUs simplifies the application design and, hence, it is considered as a part of the MLCA hardware model.

3.2 Programming Model

The hardware features of the MLCA give rise to a natural programming model that is very similar to sequential programming. The MLCA programming model is layered. The bottom layer comprises the task bodies, or simply the tasks. Each task implements a given functionality and has defined inputs and outputs. A task can be a sequential C program, a block of assembly code executing on a programmable PU such as a processor or DSP core, or a predefined functionality of a non-programmable PU such as a hardware block.

The top layer of the model is a single task-program, referred as a *control program*, which executes on the CP. It is a sequential program that specifies task instructions, and is expressed, in a C-like language called *Sarek*. The language replaces function calls with

```
int Add() {
    int n1 = readArg(0);
    int n2 = readArg(1);

    writeArg(0, n1 + n2);

    return (n1+n2) != 0 ;
}
```

Figure 3.2: An example task function body.

task calls, and adds explicit direction indications (in or out) for function arguments.

Figure 3.2 and Figure 3.3 illustrate the MLCA programming model. The task `Add` shown in Figures 3.2 is expressed as a C function that computes the sum of two integers. The function has no formal arguments. Instead, it communicates with the control program through an API, obtaining input data with a `readArg` call, and writing results using an analogous `writeArg` call. For example, `readArg(1)` reads the second input to the function, while `writeArg(0)` writes the first output of the function. The task also returns a condition code that is written to a condition register in the CP, and may be used in a control program to make control decisions.

The main part of the corresponding control program for the example is shown in Figures 3.3. It makes four calls to the task `Add`. In each call, the variable names of the inputs and outputs of each task are specified. In addition, access type indicators (in or out) are also specified for each variable.

Sarek has while-loops and if-statements for control-flow, but it has only two data types: control variables and data variables. Control variables store the return values of task calls, and are used to decide flow of control in conditionals and loops. Sarek allows bitwise logical expressions on control variables. Data variables provide input and output arguments for task calls, as illustrated in the example above. There is no restriction on the type of data that can be carried by the data variables. In that sense, a data variable may contain scalar values (such as integer) or pointer values.

Semantically, data variables that are outputted from tasks in a PU are available when

```
do {
    ...
    notzero = Add(in width1, in width2,   out totwidth);
    notzero = Add(in width3, in totwidth, out totwidth);
    if (notzero) {
        Div(in area1, in totwidth, out length1);
    }
    notzero = Add(in width4, in width5,   out totwidth)
    notzero = Add(in width6, in totwidth, out totwidth);
    if (notzero) {
        Div(in area2, in totwidth, out length2);
    }
    ...
    notfinished = NotDone(in index);
} while(notfinished);
```

Figure 3.3: A sample control program.

the task writes them using the `writeArg` call. By contrast, control variables are available to the upper layer only when a task has completed execution. Consequently, the first conditional `if (notzero) ...` in Figure 3.3 can be evaluated only after the preceding task `Add` has completed, even though the input argument `totwidth` for the following task `Div` is available earlier.

Sarek is compiled to generate a register-level intermediate representation of the program similar to assembly, which is called *HyperAssembly (HASM).* The HASM code fragment in Figure 3.4 corresponds to the control program in Figure 3.3. In this HASM code, control variables are stored in URF control registers, denoted `CRx`. Data variables are stored in URF data registers, denoted `Rx`. For URF data registers, the access type of the register is also given, as `:r` for inputs, and `:w` for outputs, which is used for dependence analysis by the hardware.

3.3 Renaming and Synchronization

In this section, we describe the URF task communication in the MLCA and describe the architecture's renaming and synchronization mechanisms.

Tasks communicate through the URF by reading from and writing to URF data

```
Do1Top:
        ...
        task Add, CR1, R5:r,R3:r,R3:w
        if false (CR1 & 0x7fffffff) jmpa If2False
If2True:
        task Div, R5:r,R3:r,R4:w
If2False:
        ...
If2End:
        task NotDone, CR2, R2:r
        if true (CR2 & 0x7fffffff)  jmpa Do1Top
```

Figure 3.4: The HASM code for the control program fragment of Figure 3.3.

registers of fixed size using the `readArg` and `writeArg` routines. However, type of data communicated through URF is not limited. Thus, both scalars and pointers can be communicated through the URF. URF is more suitable for scalar data, but it is less suitable for aggregate data such as buffers and structures because of their large sizes.

Consider the simple code segment shown in Figure 3.5 which illustrates the data communication through the URF. In the sample control program of Figure 3.5(a), `TaskB` and `TaskC` use the scalar `count` value produced by `TaskA` and, similarly, `TaskE` uses the `count` value produced by `TaskD`. Consequently, `count` is the output argument of `TaskA` and `TaskD` and the input argument of `TaskB`, `TaskC` and `TaskE`. It is important to note that, since the `count` value produced by `TaskA` is not used by `TaskD`, it is not necessary to make `count` an input argument of `TaskD`. When the control program is assembled, the HASM program in Figure 3.5(b) is obtained, after the allocation of the physical registers for the task arguments.

One of the most significant benefits of the MLCA is register renaming at the task level. At run-time, dependences among URF registers are tracked by the CP. In case of an output or an anti dependence, the CP allocates a new register for the destination of the subsequent task instruction, effectively renaming the register and resolving the dependence. This is illustrated using the example HASM program in Figure 3.5(b). The CP will rename the destination register of `TaskD`, together with the subsequent use of the

```
TaskA(out count);
TaskB(in count);
TaskC(in count);
TaskD(out count);
TaskE(in count);
```

(a) Control program with URF data communication.

```
TaskA, R1:w
TaskB, R1:r
TaskC, R1:r
TaskD, R1:w
TaskE, R1:r
```

(b) The corresponding HASM program.

Figure 3.5: Sample code fragments using URF data communication.

same register in `TaskE`, as shown in Figure 3.6. As the result of register renaming, the anti URF dependences `TaskB` δ^a `TaskD` and `TaskC` δ^a `TaskD`, and the output dependence `TaskA` δ^o `TaskD` are resolved, allowing their parallel and out-of-order execution.

The MLCA programming model intentionally lacks any synchronization routine, for simplicity. Instead, the URF and the CP are handling both the synchronization and the data communication of the tasks. Basically, when there is a URF flow dependence

```
TaskA, R1:w
TaskB, R1:r
TaskC, R1:r
TaskD, R2:w
TaskE, R2:r
```

Figure 3.6: The HASM program of Figure 3.5(b) after register renaming.

between two tasks, they are scheduled sequentially by the CP.

For instance, when the HASM program shown in Figure 3.6 is executed by the CP, after renaming is performed, `TaskB` and `TaskC` will be dispatched after `TaskA` is completed, because of the URF flow dependences `TaskA` δ^f `TaskB` and `TaskA` δ^f `TaskC`. Similarly, `TaskE` will only start after `TaskD` is completed.

3.4 Benefits of the MLCA

The combination of architecture and programming model of the MLCA give rise to a number of advantages for SoC designs:

- *Reduced software complexity.* The programming model of MLCA is close to that of sequential programming. An order of execution is given to the CP and the PUs, which when executed sequentially is considered to provide correct results. This alleviates the need for explicit parallel programming, which leads to considerable software complexity.

- *Automatic extraction of the parallelism.* Similar to instruction-level parallelism, speedup can be achieved through register renaming and out-of-order execution. For output and anti dependences, the CP allocates new registers, allowing the tasks to run in parallel on separate PUs.

- *Multiple levels of parallelism.* The MLCA combines task-level parallelism at the top-level and instruction-level parallelism in the PUs. Task-level parallelism is potentially higher than the instruction-level parallelism that can be extracted from sequential code [28].

- *Separation of communication and synchronization from computations.* Synchronization and communication are often significant contributors to the complexity and cost of an embedded system. Both can lead to contention on interconnects, increased latency, and power consumption. In addition, software development is complicated by the need to insert synchronization and communication primitives. The MLCA programming model provides separation of synchronization and communication on the one hand, and computations on the other. Computations are done entirely within the tasks, i.e. they are accomplished by the PUs. Task communication and implicit synchronization are performed by the CP and the URF. This contrasts with most current parallel programming models used with existing multi-processor SoCs, where the communication and synchronization are explicit and are mixed in with the application code modules.

- *Efficient communication.* Tasks may communicate through the URF instead of relying on a shared memory, leading to potentially faster and more efficient communication.

- *Fast architecture exploration.* The modularity and flexibility of the template architecture that accommodates heterogeneity and the independence of the code from resource management, allow fast exploration of a MLCA configuration for a specific application.

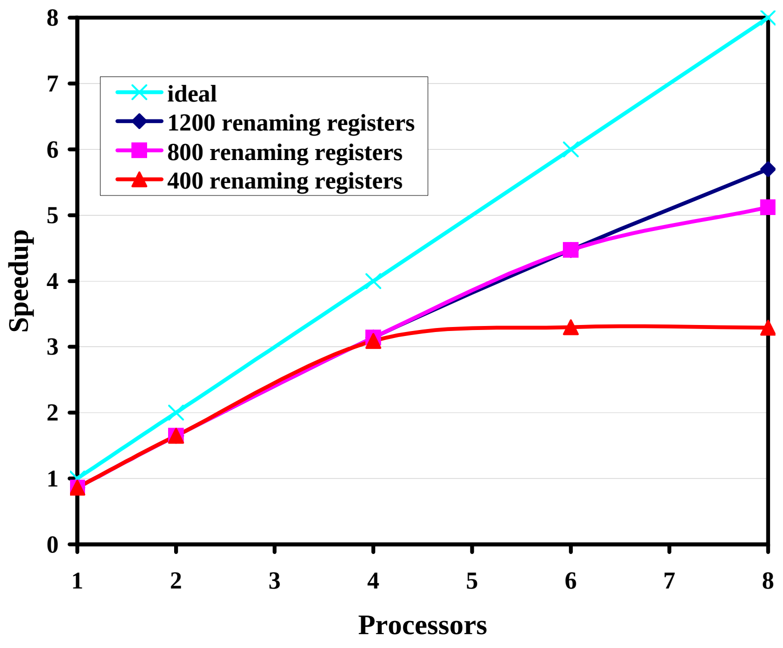
3.5 Performance

We have performed a preliminary study of the performance of the MLCA using two realistic multimedia applications and a functional simulator [20, 21]. In this section, we report on the overall performance of the these multimedia applications on the MLCA, in order to show the potential for the architecture.

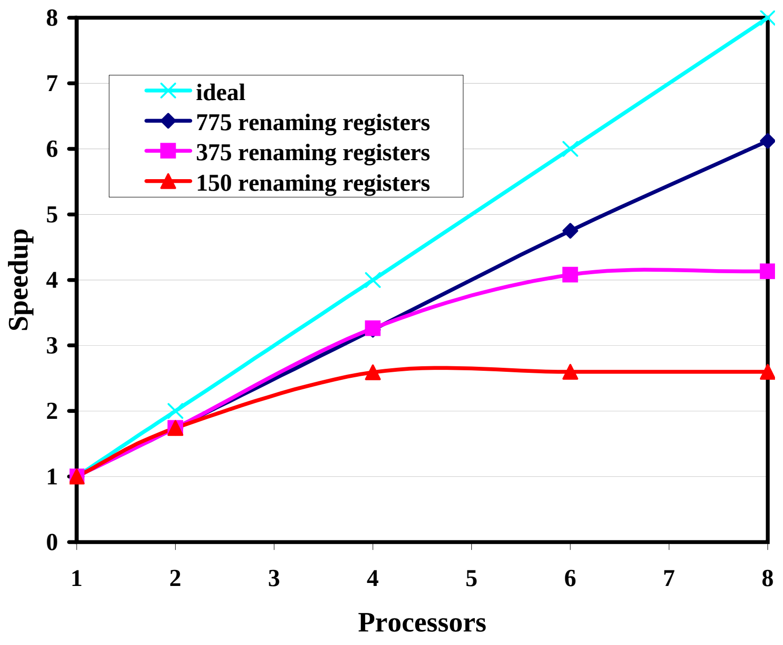
The two applications we use are MAD and FMR. MAD is an MPEG audio decoder that translates MPEG layer-3 (mp3) files into 16-bit PCM output. FMR is an audio application that performs FM demodulation on a 16-bit input data stream, producing 32-bit output data stream. Figure 3.7 shows speedups for the MAD and FMR applications as a function of both the number of processors and the number of renaming registers.

In Figure 3.7, it is seen that each application exhibits scaling speedup, which is relatively close to ideal when the number of renaming registers is sufficiently large. The figure also shows that speedup of each application depends on the number of renaming registers; in general, the more renaming registers are available, the higher is the speedup. This behavior is due to the fact that the increase in the number of renaming registers breaks more false inter-task dependences, allowing more tasks to execute in parallel. When all false dependences are removed by the addition of enough renaming registers, the execution speed of each application is dictated by the true dependences among its tasks. After enough renaming registers are used, the addition of more renaming registers does not lead to the execution of more tasks and thus further speedup.

In our view, the MLCA is a promising novel architecture providing easy portability of applications, while good and scaling performance is obtained. Its natural programming model reduces software complexity, with automatic synchronization of tasks, whereas the architecture provides both fine and coarse grain parallelism. The MLCA also relieves the programmers from the burden of the implicit synchronization and communication routines inside computation, by separating their design space.



(a) MAD



(b) FMR

Figure 3.7: The speedup of the MAD and FMR applications [20, 21].

Chapter 4

Compiler Support For The MLCA

In this chapter, we describe the compiler support for the MLCA. Section 4.1 introduces our vision of the MLCA compilation environment, intended to create high-performance HASM programs and tasks, starting from sequential applications. Section 4.2 presents performance and efficiency issues in control programs and discusses the role of the MLCA compiler in overcoming them.

4.1 The MLCA Compiler

We envision a compilation environment for the MLCA, which is shown in Figure 4.1. The environment converts sequential application code into a control program and a set of tasks that execute on an instance of the MLCA and that exhibit high performance.

The sequential application code is processed using a task selector that groups the instructions of the application in tasks and forms a control program that represent the data and control flow among these tasks. A control program together with its corresponding tasks is referred as the *task distribution.* The task distribution produced by the task selector is processed by an optimizer to improve the performance. The optimized control program is assembled to HASM by an assembler and the optimized tasks together with the other functions of the application are compiled to machine code by a native compiler

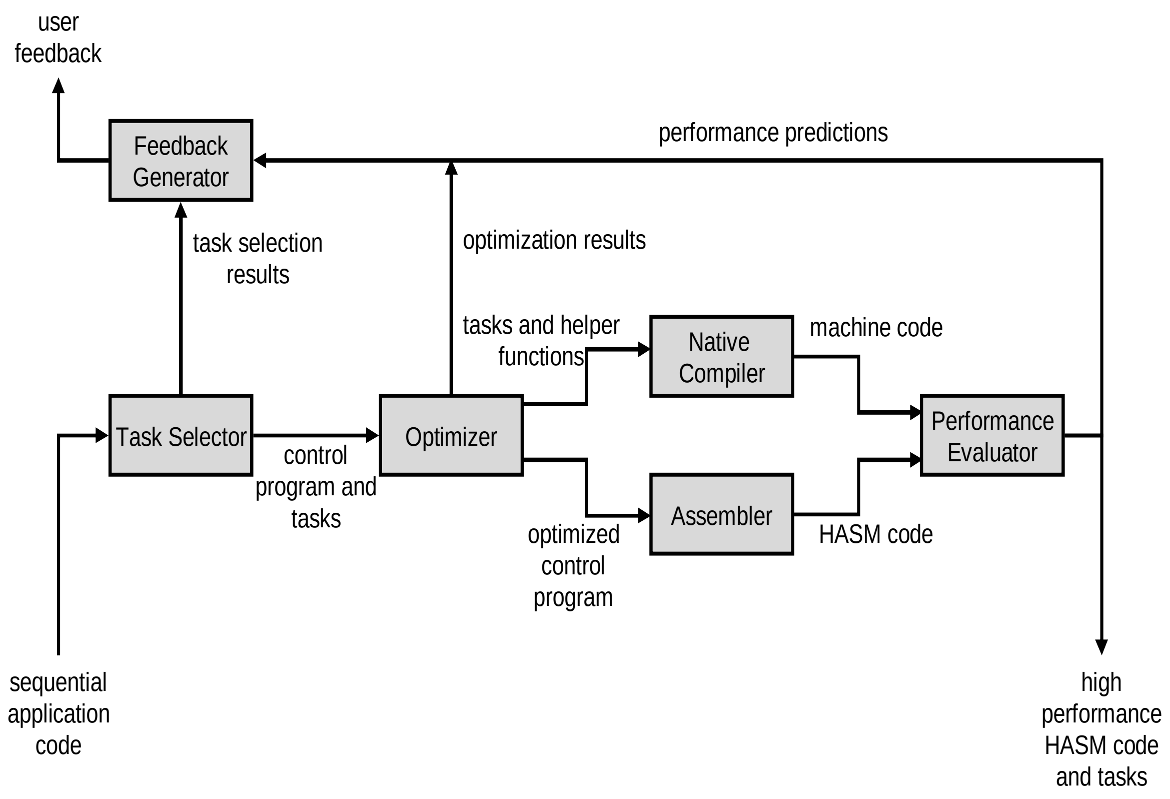


Figure 4.1: The MLCA compilation environment.

(such as `gcc`, `ecc`, etc.) for the target PUs of the MLCA instance. A performance evaluator is used to predict the performance of these optimized and assembled/compiled codes. The performance predictions, and the output of the task selector and the optimizer are used to provide feedback to the user through a feedback generator. The compilation environment is invoked iteratively, by starting with an initial task distribution, revising and optimizing the distribution in every invocation. The user decides on when to stop in this iterative process based on the feedback generated by the feedback generator and the performance requirements of the application.

We believe this compilation environment is suitable for the class of applications the MLCA is targeted for, namely multimedia applications. An MLCA instance is intended to execute one or few applications, and hence designers are willing to spend the time and effort in this iterative process to ensure that the code performs well. Further, this environment allows experimenting with many MLCA instances for designers to reduce cost/performance. These designers are usually familiar with the application to a point which allows them to put the feedback provided by the compiler into the context of the application and the MLCA instance used.

In the remainder of this section, we elaborate on each component of the compilation environment.

The task selector is responsible of determining which instructions of the application are to serve as tasks. A control program that represents the control and the data flow among the selected tasks is also generated by the task selector. This control program and the corresponding tasks are newly created in the first invocation of the task selector or they are revised in later invocations.

An important aspect in the selection of tasks is task granularity. Coarse grain tasks tend to hide parallelism by assigning large portions of sequential code on a single processor. In contrast, fine-grain but higher number of tasks increase contention over the CP, and consequently, the task issue costs. Therefore, fine grain tasks should be selected over the coarser grain ones only if there is parallelism to be extracted. However, the impact of the task granularity on the performance of control programs is hard to predict at compile-time. Consequently, the selection of the optimum task granularity is best achieved with an iterative and profile-based approach. Furthermore, such an approach can provide the optimizer with crucial information that may not be obtainable through static compiler analyses and increase the effectiveness of the optimizations. As a result, the MLCA compilation environment is chosen to be iterative and includes a performance evaluator.

The purpose of the optimizer is to solve performance and correctness issues in control programs and tasks, which will be described in the remainder of this chapter. In fact, in order to obtain sufficient performance from a task distribution, generated by the task selector, the control program and the tasks should be optimized through some transformations specific to the MLCA, which will be presented in Chapter 5. These code transformations rely on some analyses results which are obtained using static compile-time analyses.

The assembler and the native compiler are necessary for generating the executable codes that will run on the CP and the PUs respectively. Consequently, the assembler is also responsible of allocating the physical registers for the URF data and control registers of the MLCA. Similarly, the native compiler generates the machine code for the target PUs and also optimizes the tasks and the other functions of the application via regular compiler analyses.

The performance evaluator provides performance feedback to the components of the compilation environment, namely the task selector and the optimizer, and the programmer. It can be a static model of the MLCA which estimates the performance of the control program based on predictions about control decisions and I/O operations. It can also be a dynamic model of the MLCA, which actually simulates the ported application, providing more accurate run-time information.

The purpose of the feedback generator is to organize the performance, task selection and optimization results in a human-readable format that the programmer can understand. In that sense, different aspects of the performance evaluation, the revisions made to the control program and any applied and/or not-applied code transformation(s) are presented to the programmer.

The focus of our work is the optimizer. We believe that for most of the multimedia applications, making a good selection of tasks is not very difficult because of the simple control flow of such applications. However, the real challenge is in finding ways of improving the performance of a control program and the corresponding tasks, which is realized by the optimizer. In the following section, we elaborate on the need for the optimizer by describing the performance and efficiency issues in control programs and tasks.

4.2 The Need For Optimization in Control Programs

In the MLCA programming model, tasks can communicate through the URF and/or shared memory. The URF communication is described in Section 3.3. In this section, we introduce shared memory data communication and present the correctness and performance problems caused by this communication. In fact, these problems necessitate compiler optimizations in control programs.

Communication through shared memory is achieved by reading and storing data from/to the specific addresses in the shared memory. However, these addresses must be communicated among tasks, in order for these tasks to access the same data. Often data in memory is located in consecutive addresses and store aggregate data structures, such as buffers and structures. Therefore, rather than communicating the addresses of each accessed element of aggregate data, it is sufficient to communicate the start address. The individual elements (elements in buffers, fields in structures) can be accessed by adding offsets to these start addresses. In other words, when some aggregate data structure produced by a task `T1` is needed to be used by another task `T2`, `T1` stores the data to a specific region in the shared memory and copies the start address of this region to a URF register. Next, `T2` copies this specific URF register containing the start address of the region to its local stack and accesses the region using this address. When the memory region is a buffer, the start address of the buffer, i.e. the pointer to the buffer is communicated through the URF. Similarly, when the memory region is a structure, the address of the structure in the memory, i.e. the pointer to the structure, is communicated through URF.

With shared memory data communication, the URF access types (read or write) of the shared memory addresses themselves do not necessarily represent the access types (read or write) of data. In other words, whether the data in the memory is read or written, the address of the data is always read from the URF by a task. This is the case because the task needs this address to access the data in the memory. The only exception is the task that allocates/creates a memory region and, thus, assigns the address of the region for the first time, in which case, the address itself is only an output argument of the task.

Figure 4.2 depicts an example of such shared memory communication. In the control program shown of Figure 4.2(a), the access types of data are shown as comments in the control program for ease of presentation. In this control program, `TaskA`, `TaskB`, `TaskC`, `TaskD` and `TaskE` access the memory region created in `Init`, via the pointer `ptr`, which points to the start of this memory region. Since the value of `ptr` is assigned in `Init`, it is an output argument of this task. In this control program, `TaskA` and `TaskD` are writing to the memory region, whereas `TaskB`, `TaskC` and `TaskE` are only reading from it. Therefore, tasks communicate the data through the shared memory, while the pointer/address of the memory region is communicated through the URF. It can be seen that all tasks read the value of the pointer from the URF (except `Init`) but, the access type of the data in memory can not be extracted just from the control program and tasks should in fact be inspected. Figure 4.2(b) depicts the HASM program created from the control program, by defining the physical registers for the task arguments.

```
//Creates the memory region
Init(out ptr);

//Writes to the whole memory region
TaskA(in ptr);

//Reads the whole memory region
TaskB(in ptr);

//Reads the whole memory region
TaskC(in ptr);

//Writes to the whole memory region
TaskD(in ptr);

//Reads the whole memory region
TaskE(in ptr);
```

(a) Control program with shared memory data communication.

```
Init, R1:w

TaskA, R1:r

TaskB, R1:r

TaskC, R1:r

TaskD, R1:r

TaskE, R1:r
```

(b) The corresponding HASM program.

Figure 4.2: Sample code for shared memory data communication.

The communication through shared memory results in synchronization and renaming problems that require compiler assistance. These problems are described in the remainder of this section.

4.2.1 Incorrect Synchronization

A serious problem caused by the shared memory communication is incorrect synchronization. This problem is caused by the fact that, the URF access types (input/output) do not match the access types (read/write) of the data; hence, the URF dependences do not reflect the dependences caused by the accesses to the data in the shared memory. Consequently, when the CP synchronizes tasks according to the URF data dependences, shared memory dependences may be violated.

Consider the control program shown in Figure 4.2(a). Since `ptr` is only input arguments of `TaskA`, `TaskB`, `TaskC`, `TaskD` and `TaskE`, these tasks will be scheduled in parallel by the CP. However, this parallel execution will violate the data dependences, i.e. flow dependences of `TaskA` δ^f `TaskB`, `TaskA` δ^f `TaskC`, `TaskD` δ^f `TaskE`, the output dependence `TaskA` δ^o `TaskD` and the anti dependences of `TaskB` δ^a `TaskD` and `TaskC` δ^a `TaskD`. These dependences are all caused by the accesses to the shared data in memory.

A simple solution to this problem is to represent the memory dependences with URF dependences. This solution is considered as a rule of thumb and applied intuitively by MLCA programmers. In the remainder of this section, we will elaborate on this solution to the synchronization problem and discuss its weakness.

In a correct control program, the shared memory data dependences should be captured by URF data dependences. This can be achieved by explicitly representing memory data dependences with URF flow dependences, i.e. additional output and input arguments in the control program. In this way, when there is a memory data dependence between any two tasks, a URF flow dependence will also exist, forcing the CP to synchronize the two tasks. Programmers intuitively realize this by artificially making any pointer indirectly accessed in a task also an output argument of the task (in addition to being an input argument). With this approach, even though the value of the pointer itself is not modified inside the task, it is written back to the URF after the task is complete. Because any task accessing the pointer will have it as an input argument, this approach

```
//Reads from the buffer using ptr pointer
Read_Data(in ptr);

//Writes to buffer using ptr pointer
Write_Data(in ptr);
```

(a) Execution of tasks is parallel, as there is no URF dependence between them.

```
//Reads from the buffer using ptr pointer
Read_Data(in ptr, out ptr);

//Writes to buffer using ptr pointer
Write_Data(in ptr);
```

(b) Execution of tasks is serialized through artificial output argument in Read_Data.

Figure 4.3: An example of synchronization output arguments.

introduces URF flow dependences caused by the accesses to the value of the pointer and, hence, prevents the violation of any flow, output and anti memory dependences with the subsequent tasks accessing the same pointer. A pointer which is made an output argument by the programmer is named a *synchronization output argument (SOA)*.

Figure 4.3 depicts an example of an SOA, where the argument `ptr`, carrying a buffer-pointer, is made an output argument in task `Read_Data` to serialize the execution of `Read_Data` and `Write_Data` because of the shared memory anti-dependence.

Although SOAs prevent incorrect synchronization, they cause unnecessary synchronization, i.e. they introduce false dependences among tasks accessing the same pointer. This is due to the fact that making a pointer both input and output arguments of all the tasks accessing it, serializes these tasks. Even though some of these accesses overlap with memory data dependences which need to be satisfied via SOAs, some does not, serializing tasks that can indeed execute in parallel. This introduces what we refer to as *synchronization false dependences (SFDs)*.

Figure 4.4 presents a case where a SOA `ptr` creates a SFD between tasks `Read_Data1` and `Read_Data2`. Nonetheless, `ptr` is needed to be declared as output argument in

```
//Reads data from buffer ptr
Read_Data1(in ptr, out ptr);

//Reads data from buffer ptr
Read_Data2(in ptr, out ptr);

//Writes to buffer ptr
Write_Data(in ptr, out ptr);
```

Figure 4.4: An example of synchronization false dependence.

`Read_Data1` and `Read_Data2` to satisfy the anti-dependenced with `Write_Data` through shared memory.

Therefore, compiler support is needed to detect SFDs and re-arrange tasks arguments in a way that only data-dependent tasks are serialized. On the other hand, in cases that SOAs are not used to satisfy the memory dependences (usually resulting in incorrect programs), the compiler should effectively synchronize tasks according to the memory dependences, using the appropriate SOAs.

4.2.2 The Renaming Problem

When tasks produce and use data communicated only through the URF, the renaming mechanism of the CP, described in Section 3.3, is successful in resolving the false dependences among tasks. In case shared memory is also a medium of communication, the URF dependences are caused by the accesses to the value of the pointer. On the other hand, the real data dependences among tasks are caused by the accesses to data in the shared memory. Since the CP can only keep track of URF accesses, it is only able to rename the registers carrying the value of the pointers. Consequently, the MLCA renaming mechanism is ineffective in resolving false shared memory data dependences. This inability to resolve false dependences on data in memory may limit the available parallelism and thus reduce the performance.

In the sample control program shown in Figure 4.2(a), the URF true dependences, caused by the accesses to the address of the memory region, are among `Init` and the remaining tasks. However, the true data dependences, caused by the accesses to the memory region, are `TaskA` δ^f `TaskB`, `TaskA` δ^f `TaskC` and `TaskD` δ^f `TaskE`. Furthermore, there are also anti data dependences of `TaskB` δ^a `TaskD` and `TaskC` δ^a `TaskD`, and the output dependence `TaskA` δ^o `TaskD`. When the HASM program of the control program, shown in Figure 4.2(b), is executed, no renaming will be performed by the CP, because `R1`, which corresponds to Sarek variable `ptr`, is (and must be) an output argument of only `Init`. Even if the renaming was performed, `R1` would be renamed. Consequently, the anti and output data dependences that exist on the shared memory will still remain, preventing the parallel execution of `TaskA`, `TaskB` and `TaskC` with `TaskD` and `TaskE`.

Thus, compiler support is needed to resolve the shared memory output and anti dependences, in order to improve parallelism.

Chapter 5

The Code Transformations

This chapter presents the code transformations designed to efficiently run MLCA applications. These code transformations are applied on the control programs which are written in the Sarek language. Thus, they are collectively referred as the *Sarek code transformations*. The task functions may also need to be modified as a side effect of these code transformations.

Section 5.1 presents *parameter deaggregation* as a solution to the renaming problem described in Section 4.2.2. Section 5.2 introduces a *memory renaming* technique for resolving the shared memory dependences caused by buffer accesses. Section 5.3 presents *buffer renaming* which is effective in resolving synchronization false dependences mentioned in Section 4.2.1. Section 5.4 describes *code hoisting* which enables a task to write its arguments earlier so that any task waiting for these arguments can start earlier.

5.1 Parameter Deaggregation

We propose parameter deaggregation as a compiler solution to the renaming problem, described in Section 4.2.2, specifically for task arguments of type pointer-to-structure. The purpose of parameter deaggregation is to expose the elements of structures in the parameter list of tasks, effectively transforming shared memory dependences to URF dependences. Consequently, the CP is able to rename the fields of the structures, which become the parameters of tasks, eliminating the false dependences for the scalar fields of these structures.

Parameter deaggregation is performed by replacing pointers to structures by the fields of the structures, until all task parameters are of primitive types (e.g., `int`, `float`, `int *`, etc.). Since structures may contain other structures or pointers to other structures in the shared memory, parameter deaggregation is a recursive transformation.

A naive replacement of task arguments of type pointer-to-structure with all the fields of the corresponding structures can result in unnecessary dependences among tasks, thus eliminating parallelism. Therefore, it is essential to consider the accesses to the structure fields in tasks during the process of deaggregation. A field of a structure should be transformed into an input argument to a task only if it is used (before being written) by the task. Similarly, a field of a structure should be made an output argument to a task only if it is written to in the body of the task. Thus, tasks should not have as input any field that they do not use. Similarly, tasks should not have any field as output that they do not define.

Parameter deaggregation is always legal because it does not change the data accesses of the program. It only transforms shared memory data dependences to URF data dependences.

Figure 5.1 depicts an example of parameter deaggregation for parameter `st_ptr`. For simplicity, accesses to structure fields are shown as comments to the task calls. Furthermore, `st_ptr` is both input and output to the tasks in order to prevent the data dependence violations, with SOAs. The structure declarations, used in the task bodies, are given in Figure 5.1(a). In fact, `st_ptr` is a pointer to `storage` structure in the bodies of the `Read_Frame_1`, `Process_Frame` and `Read_Frame_2` tasks. In the control program of Figure 5.1(b), `Read_Frame_1` defines `count` and `value` fields of the `storage` structure, thus, parameter deaggregation makes these fields output arguments to the task. On the other hand, task `Process_Frame` uses `value` field of the structure, which is an input parameter of the task after deaggregation. `Process_Frame` task also defines the `result` field of the `outcome` structure which is indirectly accessed with the `output` field of type pointer-to-structure. Recursive parameter deaggregation makes this `result` field an output parameter. Furthermore, `Read_Frame_2` task is also affected by the transformation, in which `value` and `count` fields are made output parameters. On the other hand, `dummy` field of the `storage` structure and `previous_result` field of the `outcome` structure are not made input/output arguments for any task as they are not read/written in any of the task bodies.

The resulting control program after parameter deaggregation is shown in Figure 5.1(c). In this control program, CP is able to eliminate the false output dependence between `Read_Frame_1` and `Read_Frame_2` tasks, by renaming the `st_ptr_count` variable, consequently executing `Read_Frame_2` task in parallel with `Read_Frame_1` and `Process_Frame`.

Parameter deaggregation has the overhead of increased contention over the URF, due to higher number of task parameters. A URF with more registers is also needed. Furthermore, each element of the structure would have to be read/written using a separate `readArg` / `writeArg` routine, which will create additional overhead in task bodies. However, structures usually contain few fields in many multimedia applications and, only few of the fields are accessed in each task. Consequently, the overhead of parameter deaggregation is expected to be low for most multimedia applications. Furthermore, the parameter deaggregation is very effective in extracting parallelism, as will be shown in Chapter 8. Thus, overhead of parameter deaggregation is expected to be outweighed by gains in speedup.

```
struct storage {
   int count;
   int value;
   struct outcome *output;
   int dummy;
};

struct outcome {
   int result;
   int previous_result;
}
```

(a) Structure definitions.

```
//Defines st_ptr->count and st_ptr->value
Read_Frame_1(in st_ptr, out st_ptr);

//Uses st_ptr->value and defines st_ptr->output->result
Process_Frame(in st_ptr, out st_ptr);

//Defines st_ptr->count and st_ptr->value
Read_Frame_2(in st_ptr, out st_ptr);
```

(b) Control program before deaggregation.

```
Read_Frame_1(out st_ptr_count, out st_ptr_value);

Process_Frame(in st_ptr_value, out st_ptr_output_result);

Read_Frame_2(out st_ptr_count, out st_ptr_value);
```

(c) Control program after deaggregation.

Figure 5.1: Parameter deaggregation example.

5.2 Memory Renaming

In this section, we describe the memory renaming technique intended to solve the renaming problem discussed in Section 4.2.2, for the task arguments of type pointer-to-buffer. The purpose of memory renaming is discussed, and two code transformations that apply memory renaming are presented.

Memory renaming allocates extra storage in memory to break memory false dependences in the same way extra registers are used in the URF to break register false dependences. Memory renaming is performed for memory buffers but not for structures because structures often have a small number of fields, and it is possible and indeed easier to rename them using the URF after deaggregation. In contrast, memory buffers have many elements that are hard to store all in URF and, even if they are stored high communication costs will be resulted for reading and writing task arguments.

In the following sections, we describe two code transformations that apply memory renaming in different ways for different data dependence situations.

5.2.1 Buffer Replication

Buffer replication is the general application of memory renaming, where an anti dependence on the shared memory is resolved for two tasks, enabling their parallel execution. With buffer replication, the memory buffer causing the memory dependence between two tasks is replicated by allocating an extra buffer and initializing this newly created buffer with the original buffer. This new copy of the buffer is given to the task that reads the buffer, while the other task that writes to the buffer still processes the original buffer. In this way, the two tasks accesses different buffers, eliminating the false dependence between them.

Buffer replication is implemented with the creation of three helper tasks. The `Init` task allocates the needed extra storage and inserts the pointer value into its output register variable. This register variable is given to the task that needs the extra storage, as input argument by replacing/renaming its original input register variable. The Copy task initializes the extra storage with the data from the original storage, in order to preserve data-flow correctness. Finally, when the extra storage is no longer needed, it is deallocated with the Finish task.

Buffer replication is always legal, because false dependences (caused by register accesses or memory accesses) can always be resolved with renaming the storage (register or memory).

Figure 5.2 depicts an example of the buffer replication where the shared memory anti-dependence between tasks Read_Data and Update_Data through buffer buf is resolved. For simplicity, the buffer regions accessed by each task are shown as comments to corresponding task calls. In the example control program, after buffer replication, task Init allocates a new copy of the buf, i.e. new_buf, and Copy initializes new_buf with the data in buf. New buffer new_buf is given to the first task of memory anti-dependence (Read_Data), while the second task (Update_Data) takes the original buffer, i.e. buf. After the new_buf is used in Read_Data, it is deallocated with task Finish.

Buffer replication has an overhead of allocating and initializing the copy of the original buffer space. The performance gain comes with the parallel execution of two tasks, which are serial without buffer replication. Therefore, buffer replication should be performed in a conservative way, by inspecting the available gain against the possible overhead.

5.2.2 Buffer Privatization

Buffer privatization resolves the memory false dependences between collections of tasks, enabling their parallel execution.

With buffer privatization, a set of tasks T_1, T_2, ..., T_n that access a memory buffer buf is divided into disjoint subsets of tasks S_1, S_2, ..., S_k that access the same data in buf. In other words, in a subset S_i, for every use of data stored in buf, the definition

```
//Writes to buf[0:10]
Write_Data(in buf, out buf);

//Reads from buf[0:10]
Read_Data(in buf, out buf);

//Writes to buf[0:10]
Update_Data(in buf, out buf);
```

(a) Control program before buffer replication.

```
//Writes to buf[0:10]
Write_Data(in buf, out buf);

//new_buf is allocated
Init(out new_buf);

//new_buf is initialized with buf
Copy(in new_buf, in buf,
     out new_buf, out buf);

//Reads new_buf[0:10]
Read_Data(in new_buf, out new_buf);

//new_buf is deallocated
Finish(in new_buf);

//Writes to buf[0:10]
Update_Data(in buf, out buf);
```

(b) Control program after buffer replication.

Figure 5.2: Buffer replication example.

of the data is also in S_i. A single private memory buffer, the same size as `buf`, is given to each subset of tasks instead of `buf`, so that tasks in each subset access a different buffer in the memory. Since, tasks that are accessing the same data are still using the same buffer, no true dependences are violated, maintaining correct execution. Since the described code transformation effectively privatizes a buffer for a set of tasks, it is named buffer privatization.

Buffer privatization can be achieved by applying a similar memory renaming code transformation scheme to the one explained in Section 5.2.1 for buffer replication; however, in this case, the scope of the memory renaming is a collection of tasks, rather than a single task. In addition, since no flow-dependence is broken by buffer privatization, renamed memory does not need to be initialized after it is created. In order to privatize a buffer `buf` for a collection of tasks T_1, T_2, ..., T_n, first, a private copy `pri` of `buf` is created with an `Init` task before any of T_1, T_2, ..., T_n starts executing. Next, T_1, T_2, ..., T_n are given the private copy by means of replacing `buf` with `pri` in the input/output arguments. Finally, after T_1, T_2, ..., T_n are all executed, the no longer needed private copy is destroyed with the `Finish` task.

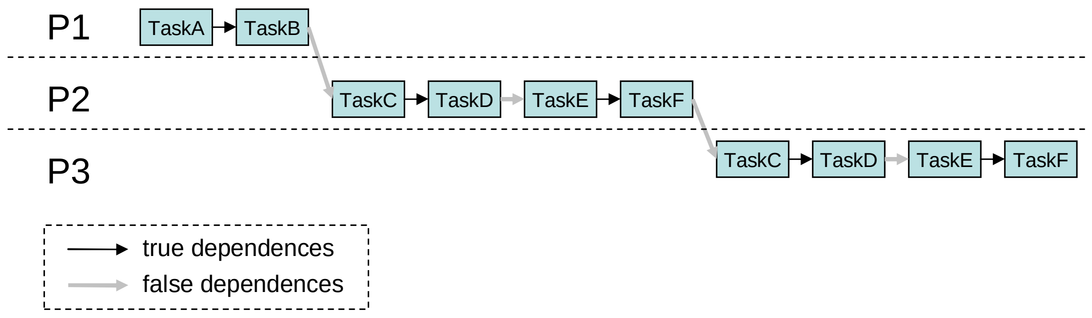
Buffer privatization is illustrated using the example control program in Figure 5.3(a). In the control program, `TaskA`, `TaskB`, `TaskC`, `TaskD`, `TaskE` and `TaskF` access a buffer `buf` in the shared memory. For simplicity, the buffer regions accessed by each task are shown as comments to corresponding task calls. Since the tasks of the control program access the same addresses in `buf`, false dependences serialize the execution of the tasks as shown in Figure 5.3(b), for three processors and two iterations of the loop.

The tasks that access `buf` can be divided into three sets S_1, S_2 and S_3 according to the data they access in `buf`, as shown in Figure 5.4. In the figure, `TaskB` is in the same set S_1 as `TaskA` because it uses the data defined in `TaskA`. Similarly, `TaskC` and `TaskD` form the set S_2, and `TaskE` and `TaskF` form the third set S_3.

Each of the three sets can be assigned a private buffer which breaks the false depen-

```
// Allocates buf of type int(*)[10]
Allocate(out buf);

// Defines buf[0:9]
TaskA(in buf, out buf);

// Uses buf[0:9]
TaskB(in buf, out buf);

while(...)
{
  // Defines buf[0:9]
  TaskC(in buf, out buf);

  // Uses buf[0:9]
  TaskD(in buf, out buf);

  // Defines buf[0:9]
  TaskE(in buf, out buf);

  // Uses buf[0:9]
  TaskF(in buf, out buf);
}

// Deallocates buf
Deallocate(in buf);
```

(a) Example control program.



(b) Tasks are serialized due to false dependences.

Figure 5.3: Example control program and run-time execution of its tasks.

$$S_1 = \{ \texttt{TaskA}, \texttt{TaskB} \}$$

$$S_2 = \{ \texttt{TaskC}, \texttt{TaskD} \}$$

$$S_3 = \{ \texttt{TaskE}, \texttt{TaskF} \}$$

Figure 5.4: The data access sets for the tasks of the running example.

dences `TaskB` δ^a `TaskC`, `TaskD` δ^a `TaskE`, `TaskF` δ^a `TaskC`, `TaskA` δ^o `TaskC`, `TaskA` δ^o `TaskE` and `TaskC` δ^o `TaskE`. Privatization in the control program is achieved with the help of `Init` and `Finish` tasks, which are creating and destroying the private buffers for the tasks of $\mathtt{S}_1$, $\mathtt{S}_2$ and $\mathtt{S}_3$. After the privatization, the `buf` arguments of the tasks are renamed to `pri`, in order to access the private buffers rather than the original buffer `buf`. After the privatization, the resulting control program is depicted in Figure 5.5.

Buffer privatization enables two types of parallelism, when a buffer `buf`, accessed by a set of tasks $\mathtt{S} = \mathtt{T}_1, \mathtt{T}_2, ..., \mathtt{Tn}$, is privatized for a subset $\mathtt{S}_p$ of `S`, that contains data dependent tasks. First, if all the tasks of $\mathtt{S}_p$ are inside a loop l, `Init` and `Finish` tasks are also inserted to the body of l. This results in the creation and the destruction of a private buffer referred by a task argument `pri` in every iteration of l. When the CP renames `pri` in each iteration of l, it will rename it along with the corresponding private buffer in the shared memory. Consequently, tasks of $\mathtt{S}_p$ are assigned a private copy of `buf` in every iteration of l, resolving loop-carried false dependences and enabling parallel execution of task instances in different iterations of l. Since the parallelism happens in a loop, we name this kind of parallelism *loop-level parallelism.*

Second, after buffer privatization, since the elements of $\mathtt{S}_p$ access a different buffer from the subset of the remaining elements $\mathtt{S}_r$ of `S`, such that $S_r = S - S_p$, the tasks in $\mathtt{S}_p$ can execute in parallel with the tasks in $\mathtt{S}_r$. Since the parallelism happens between the tasks in the same body of a loop or in the body of the main program, we name this kind of parallelism *body-level parallelism.*

Depending on where the private copy is created or destroyed, privatization can happen in the main body of the control program or in a loop body. In the first case, only body-level parallelism can be obtained, as the private buffer is created and destroyed once in the main body of the control program. However, in the latter case, both body-level and loop-level parallelism is possible, as the tasks that access the private buffer `pri` of `buf` can execute in parallel with each other as well as the remaining tasks in the loop.

```
// Allocates buf of type int(*)[10]
Allocate(out buf);

// Allocates pri of type int(*)[10]
Init(out pri);

// Defines pri[0:9]
TaskA(in pri, out pri);

// Uses pri[0:9]
TaskB(in pri, out pri);

// Deallocates pri
Finish(in pri);

while(...)
{
  // Allocates pri of type int(*)[10]
  Init(out pri);

  // Defines pri[0:9]
  TaskC(in pri, out pri);

  // Uses pri[0:9]
  TaskD(in pri, out pri);

  // Deallocates pri
  Finish(in pri);

  // Allocates pri of type int(*)[10]
  Init(out pri);

  // Defines pri[0:9]
  TaskE(in pri, out pri);

  // Uses pri[0:9]
  TaskF(in pri, out pri);

  // Deallocates pri
  Finish(in pri);
}

// Deallocates buf
Deallocate(in buf);
```

Figure 5.5: The control program after buffer privatization for the running example.

In the running example, when the control program shown in Figure 5.5 is taken by the CP, for each `Init` task, the `pri` output argument will be renamed at run-time, together with the `pri` input arguments in the subsequent tasks, as shown in the Figure 5.6. However, each renamed argument `arg1`, `arg2` and `arg3` will represent a new buffer in the memory. Since the tasks of the control program are accessing distinct regions in the shared memory, no false dependence exists and the tasks will execute in parallel as shown in Figure 5.7 for five processors and two iterations of the loop. The tasks of sets $\mathtt{S}_1$, $\mathtt{S}_2$ and $\mathtt{S}_3$ are executing in parallel with each other inside the loop and the body of the main program, which realizes the body-level parallelism. On the other hand, tasks of $\mathtt{S}_1$, as well as $\mathtt{S}_2$, are executing in parallel with each other in different iterations of the loop, which creates loop-level parallelism. The existing type of parallelism among the processors is shown in Figure 5.8.

In order for the buffer privatization to be legal for a buffer `buf`, it is necessary that, after buffer privatization, no true dependences are broken among the tasks that are accessing `buf`. When `buf` is privatized for a set of tasks `S`, since `S` contains all data-dependent tasks, no true dependence exists with the remaining tasks. As a result, body-level parallelism will always be valid. However, when loop-level parallelism is possible, i.e. tasks of `S` are in a loop, the dependences among the tasks of `S` may be loop-carried. In such cases, private copies created in each iteration of the loop will break these loop-carried true dependences, causing incorrect execution. Therefore, in order for the buffer privatization to be legal for a buffer `buf` and a set of tasks `S`, no loop-carried flow-dependences should exist between the tasks of `S` caused by accesses to `buf`. In fact, in the buffer privatization example, the data access sets shown in Figure 5.4 can be privatized because the tasks they contain do not have loop-carried dependences with each other.

Buffer privatization enables both body-level parallelism and loop-level parallelism. As it is mentioned in Section 2.9, a similar well-known compiler transformation, called array privatization, privatizes arrays for iterations of loops, which are the units of parallelism.

```
// Allocates buf of type int(*)[10]
Allocate(out buf);

// Allocates arg1 of type int(*)[10]
Init(out arg1);

// Defines arg1[0:9]
TaskA(in arg1, out arg1);

// Uses arg1[0:9]
TaskB(in arg1, out arg1);

// Deallocates arg1
Finish(in arg1);

while(...)
{
  // Allocates arg2 of type int(*)[10]
  Init(out arg2);

  // Defines arg2[0:9]
  TaskC(in arg2, out arg2);

  // Uses arg2[0:9]
  TaskD(in arg2, out arg2);

  // Deallocates arg2
  Finish(in arg2);

  // Allocates arg3 of type int(*)[10]
  Init(out arg3);

  // Defines arg3[0:9]
  TaskE(in arg3, out arg3);

  // Uses arg3[0:9]
  TaskF(in arg3, out arg3);

  // Deallocates arg3
  Finish(in arg3);
}

// Deallocates buf
Deallocate(in buf);
```

Figure 5.6: CP renames the `pri` arguments along with private buffers in the shared memory.

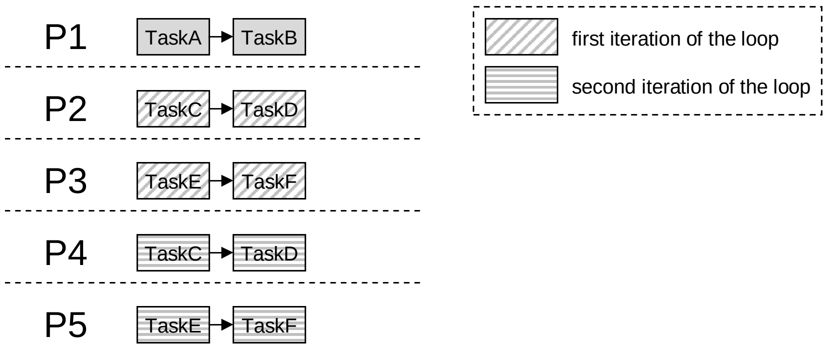


Figure 5.7: The run-time execution of tasks after buffer privatization for the running example.

```
P1 & P2 & P3 : body-level parallelism
P1 & P4 & P5 : body-level parallelism
P2 & P4 : loop-level parallelism
P3 & P5 : loop-level parallelism
```

Figure 5.8: The types of the available parallelism among processors after buffer privatization.

As a result, loop-level parallelism among the loop body instructions is obtained. Buffer privatization, on the other hand privatizes buffers for tasks, which are the units of parallelism in a control program. Thus, compared to array privatization, buffer privatization offers parallelism of finer granularity.

Buffer privatization has the overhead of creating and destroying private buffers. Therefore, buffers should be privatized for a set of tasks `S`, if either body-level or loop-level parallelism is obtainable. In other words, when accesses to buffers in the memory or to URF serialize tasks of `S` with each other in loops and with the remaining tasks of the control program, a buffer should not be privatized.

Although buffer privatization is effective in eliminating false dependences, since its application depends on some legality conditions, it does not guarantee to eliminate all the false memory dependences that exist in a control program. Thus, buffer replication code transformation, which aims to resolve false dependences among every task pairs, is necessary and may potentially resolve the false dependences that can not be resolved with buffer privatization.

5.3 Buffer Renaming

Buffer renaming is a solution to the synchronization false dependences problem, discussed in Section 4.2.1. When two tasks have a SFD, the dependence creating SOA is renamed, eliminating the false dependence. In order to ensure dependences with other tasks, i.e. to control synchronization, artificial URF dependences are created with temporary variables.

Figure 5.9 illustrates buffer renaming with an example control program. In the example, there is a SFD between tasks `Read_Data1` and `Read_Data2`. Buffer renaming renames the SOA in task `Read_Data1` (it might as well be `Read_Data2`) with an artificial register variable `arti`. This artificial variable is given to task `Write_Data` as an input parameter, in order to synchronize `Read_Data1` and `Write_Data` tasks in serial by satisfying the

```
//Reads data from buffer buff
Read_Data1(in buff, out buff);

//Reads data from buffer ptr
Read_Data2(in buff, out buff);

//Writes to buffer ptr
Write_Data(in buff, out buff);
```

(a) Control program before buffer renaming.

```
//Reads data from buffer buff
Read_Data1(in buff, out arti);

//Reads data from buffer ptr
Read_Data2(in buff, out buff);

//Writes to buffer ptr
Write_Data(in buff, out buff, in arti);
```

(b) Control program after buffer renaming.

Figure 5.9: Buffer renaming.

memory anti-dependence with an artificial URF true dependence.

Buffer renaming has the overhead of using extra register variables, which may require a bigger URF. Consequently, buffer renaming should be performed conservatively, if it enables parallel execution or when the URF size is not a concern.

5.4 Code Hoisting

Code hoisting is applied to the body of a task `T` in order to move `writeArg` calls to the earliest execution location possible. This enables other tasks, which are dependent to task `T` to possibly start executing earlier.

For hoisting scalar arguments (`int`, `char`, `long`, etc), we follow the following algorithm. Using standard compiler analyses[1], for each `writeArg` call for an argument `arg`

[1]A very busy expressions algorithm is used [23].

of the task, we find the earliest point `P` in the program, referred as an *hoisting point*, beyond which the `writeArg` is always called and beyond which a definition of `arg` does not exist. If there exists such a hoisting point `P` for a `writeArg` routine, the `writeArg` call is moved to `P`. On the other hand, if such a hoisting point `P` does not exist, the `writeArg` call remains in its original location in the program. In case the hoisting point `P` is same for multiple `writeArg` calls of the same argument, only one `writeArg` call is moved to `P` and the other ones are removed from the program. Finally, in case a `writeArg` call is already is in its hoisting point, it remains in its position in the program. In order not to cause missing or extra calls to `writeArg` routines, hoisting points are not chosen to be inside loops or if statements[2].

For hoisting pointer arguments (`int *`, `int (*)[]`, etc), we follow the same hoisting algorithm of the scalar arguments, but we use a different definition of a hoisting point. In order not to break synchronization false dependences, for arguments of type pointer, we define the hoisting point as the program point beyond which a `writeArg` for the argument is always called and the argument is not accessed (used or defined). By considering the accesses to the pointer arguments, rather than the definitions, we guarantee that the arguments will not be written to the URF before they are accessed.

Figure 5.10 depicts an example of code hoisting. In the sample task function shown in Figure 5.10(a), the program points of concern are represented as `P1` and `P2`; whereas the `writeArg` calls are indexed as `W1`, ..., `W8`. `W3` and `W7` write the `size` output argument of the task. For these calls, `P1` is the hoisting point, i.e. the point beyond which no definition of `size` exists and both `W3` and `W7` are always called. Therefore, code hoisting algorithm moves one of the calls to `P1` and removes the other one from the program. `W5` writes the `count` output argument of the task. For `W5`, `P2` is the hoisting point (beyond `P2` no definition to `count` exists and `W5` is always called). Thus, `W5` is moved to `P2`. For `W1`, `W2` and `W4` no hoisting point exists, therefore, they remain in their original position

[2]A hoisting point should dominate its corresponding `writeArg` call.

in the program. `W6` and `W8` are already in their hoisting points, because the scalar output variable `output` is last defined just before `W6` and the pointer output variable `input` is last accessed just before `W8`. Figure 5.10(b) shows the resulting task function after code hoisting is applied.

```
int Task_Function() {
  int count, size, output = 0;
  int (*input)[1000];

  input = readArg(0);
  count = readArg(1);
  size = readArg(2);

  size += 10;
  //P1
  if(count > 1000)  {
     writeArg(0, count);  //W1
     writeArg(1, output); //W2
     writeArg(2, size);   //W3
     writeArg(3, input);  //W4
     return 0;
  }
  count++;
  //P2
  for(int i = 0; i < size; i++)
    output += (*input)[i];

 writeArg(0, count);  //W5
 writeArg(1, output); //W6
 writeArg(2, size);   //W7
 writeArg(3, input);  //W8
 return 1;
}
```

(a) Task function before code hoisting.

```
int Task_Function() {
  int count, size, output = 0;
  int (*input)[1000];

  input = readArg(0);
  count = readArg(1);
  size = readArg(2);

  size += 10;
  writeArg(2, size);

  if(count > 1000)  {
     writeArg(0, count);
     writeArg(1, output);
     writeArg(3, input);
     return 0;
  }
  count++;
  writeArg(0, count);

  for(int i = 0; i < size; i++)
    output += (*input)[i];

 writeArg(1, output);
 writeArg(3, input);
 return 1;
}
```

(b) Task function after code hoisting.

Figure 5.10: Code hoisting example.

Chapter 6

Transformation Algorithms

This chapter presents a detailed description of the parameter deaggregation, buffer privatization, buffer replication and buffer renaming code transformations, discussed in Chapter 5. Section 6.1 presents the preliminary analyses performed in preparation for all four code transformations. Section 6.2 introduces the concept of pointer webs and discusses the algorithm to find them. Section 6.3 describes the algorithm for parameter deaggregation. Section 6.4 presents the section data flow solver which is useful in computing data dependences among task calls caused by the accesses to memory buffers. Section 6.5 discusses various graph representations for task data dependences. Section 6.6 presents the stages of buffer privatization code transformation. Section 6.7 discusses the algorithm for buffer replication. Section 6.8 describes the buffer renaming code transformation.

6.1 Preliminary Analyses

There are a number of preliminary compiler analyses performed on the input control program in preparation for our compiler transformations. They comprise a number of standard analyses and are described in the remainder of this section.

First, the control flow graph (CFG) of the input control program is constructed. In this CFG, we elect to make each task call a basic block of its own in order to simplify the analyses. The dominator, post-dominator, depth-first traversal order, the ancestor(s) and the descendant(s) of each basic block are determined using this CFG [23].

Second, the reaching definitions for task arguments, which are of data type `reg_t` in the control program, are computed using the CFG and a standard forward any-path data flow analysis. In this data flow analysis, each task argument is treated as a scalar and marked as definition/use based on its access type, i.e. output/input. Finally, the def-use and use-def chains are formed for task arguments, using the reaching definitions and standard compiler analyses [23].

The Sarek code transformations that will described in the remainder of this chapter rely on some analyses results of the task functions. These results are provided by a C compiler in the context of the MLCA Optimizing Compiler, as it will be described in Chapter 7. However, when the algorithms of the code transformations are presented, the analyses results of the task functions are shown as comments in the example control programs, for simplicity.

6.2 Pointer Webs

Arguments to tasks can be of two types: *scalar values* and *pointers*. Further, pointer arguments may be *pointers to arrays* or *pointers to structures*. It is important to point out that, because Sarek lacks strong typing (all variables are of type `reg_t`), the same Sarek variable may be of different types in different tasks of the control program. Since the code transformations are analyzing buffers and structures in the memory, it is crucial to identify the task arguments of type pointer and determine their pointed data type, i.e. buffers and structures.

In addition, a Sarek variable of type pointer can carry different pointer values throughout the execution of a control program. Thus, a Sarek variable, pointing to a buffer/structure in a task, may point to a different buffer/structure in another task. Further, two Sarek variables may point to the same buffer/structure in the memory, because of aliasing. As a result, it is not possible to exactly represent buffers and structures that exist in the memory by just Sarek variables. Since the code transformations are making use of buffers and structures, it is important to identify which buffer/structure each task argument is referring to.

We define *a pointer web* as the list of task arguments that are referring (pointing) to the same buffer or structure in the memory. Pointer webs are named as *buffer webs* or *structure webs* depending on the type of the referred data. Each element of a pointer web is of type buffer/structure pointer inside the task it appears in and points to the same buffer/structure. In that sense, each pointer web represents a single buffer or structure that exists in the memory during the execution of the control program. As a result, a control program has as many pointer webs as the number of the dynamic buffer/structure allocations escaping task functions[1]. Arguments in pointer webs are represented with the basic block id (bb_id) of their respective task call and their argument id (arg_id) in that task call, which starts from zero. In every code transformation and analyses using buffers and structures of the shared memory, pointer webs will be used to represent the buffers and structures in the control program.

Pointer webs are similar to register webs used for register allocation [23]. However, they differ in some key aspects. First, a register web starts with the allocation of a register in the register file, which happens in every definition of a variable. In contrast, a pointer web starts with the allocation of the buffer/structure in the memory, which does not happen every time a Sarek argument is written to. In other words, unlike register webs, a definition of a Sarek variable represents a new pointer web, *only if* the defined Sarek variable refers to a newly created/allocated memory region, inside a task.

Second, a variable does not necessarily need to be defined in a task to be an output

[1]Temporary buffers or structures used in task functions have no affect on the overall execution of the control program as the data they contain is not accessed in the other tasks.

```
TaskA(out buf1);

TaskB(in buf1, out buf2);

TaskC(in buf2);
```

(a) Sample control program.

```
int Function_of_TaskB()
{
  int (*var)[10];

  var = readArg(0);
  writeArg(0, var);

  return 1;
}
```

(b) Task function of `TaskB`.

Figure 6.1: Example control program and task function.

argument of this task. In other words, output arguments of a task can sometimes carry the same data as the input arguments of the same task, even if they are different Sarek variables in the control program. In that sense, an output argument of a task may be in the same pointer web as an input argument, even if they are not the same Sarek variables. Figure 6.1 depicts such a case, where the Sarek variables `buf1` and `buf2` of the control program (Figure 6.1(a)) refer to the same structure in the memory, because local variable `var` is written to URF without any modification in the task body of the `TaskB` task (Figure 6.1(b)).

Thus, a pointer web starts with the allocation of a buffer and/or a structure in the memory and includes all the accesses (both uses and definitions) to the buffer and/or the structure throughout the control program and may consist of more than one Sarek variable. Consequently, pointer webs are computed using def-use chains for task arguments and the results of task function analyses including allocation/deallocation, types of input/arguments and definition of arguments.

```
Find_Pointer_Webs:

for each task call task_call in the control program
  for each output argument arg in the task_call
    if arg is a pointer to buffer/structure in task function
      if arg is allocated in the task function of task_call
        Create a new buffer/structure web ptr_web
        Insert arg (arg_id of arg + bb_id for task_call) to ptr_web
        Fill_Pointer_Web ptr_web starting from arg

Fill_Pointer_Web ptr_web starting from start_arg:

for each input argument in_arg that start_arg is reaching
  Insert in_arg (arg_id + bb_id) to ptr_web
    if in_arg is not deallocated in task function
      if in_arg is also an output argument out_arg of task call
        if in_arg is defined in task function
          mark ptr_web as not-optimizable
        Insert out_arg (arg_id of arg + bb_id for task_call) to ptr_web
        Fill_Pointer_Web ptr_web starting from out_arg
```

Figure 6.2: Algorithm to find pointer webs.

If a pointer is re-defined in a task but not allocated, this means that the pointer no longer points to the start of the buffer or structure it was pointing. Thus, in such cases, the algorithm marks the pointer web containing this pointer as not-optimizable for the Sarek code transformations. For simplicity, aliasing between task arguments is not included in the algorithm. However, the support for aliasing is simple to incorporate to pointer webs. If aliasing analysis of the task functions proves that two output arguments of a task point to the same memory location, they should be inserted to the same pointer web. If analyses can not prove that two pointers are not aliases, the pointer webs which include these two pointers should not be processed by the code transformations, for conservativeness.

The algorithm to find pointer webs is depicted in Figure 6.2. Figure 6.3 illustrates pointer webs with a sample control program. In the control program shown in Figure 6.3(a), for simplicity, the analyses results of the task functions are represented with comments to the task calls. In the control program, `Init` task allocates a buffer of type `int(*)[10]` and writes the pointer of this buffer to `buf1` Sarek argument, as its 0^{th} output argument. `TaskA` takes the `buf1` argument as input, accesses the buffer and outputs

```
// Allocates the int(*)[10] buffer buf1
Init(out buf1);

// buf1 is outputted without being defined
TaskA(in buf1, out buf1);

// buf3 carries the value of buf1
TaskB(in buf1, out buf3);

TaskC(in buf3);

// buf3 is deallocated
Finish(in buf3);
```

(a) Sample control program.

```
Buffer Web 1 = { (Init, 0),
                 (TaskA, 0),
                 (TaskA, 1),
                 (TaskB, 0),
                 (TaskB, 1),
                 (TaskC, 0),
                 (Finish, 0) }
```

(b) Pointer webs.

Figure 6.3: Pointer webs example.

the pointer, without modifying it, as its 0^{th} output argument back to the `buf1`. `TaskB` takes `buf1`, assigns it to a new variable and outputs the new variable to `buf3`, as its 0^{th} output argument. `TaskC` gets `buf3` and accesses the buffer without outputting any argument. `Finish` task takes `buf3` and deallocates the buffer.

In this example control program, each task is called once. Thus, for simplicity, we refer to each task call with the name of the task, although in general the name does not represent a single task call. Further, each input and output argument will be referenced by a unique id starting from zero, which includes all the input and output arguments of the task. In other, words, the representation `(T, n)` signifies the n^{th} argument of task `T`.

In the example control program of the figure, `Init` allocates a buffer, thus a new buffer pointer field entry is created. `(Init, 0)`, which carries the pointer to the allocated buffer, is added to this buffer field. Furthermore, since `(Init, 0)`, reaches to

`(TaskA, 0)`, `(TaskA, 0)` is also added to the pointer web. `(TaskA, 0)` is written back to URF as `(TaskA, 1)` without being modified, thus `(TaskA, 1)` is also inserted to the pointer web. Next, the reaching input argument for `(TaskA, 1)`, which is `(TaskB, 0)`, is added. `(TaskB, 1)` carries the same value of the pointer as `(TaskB, 0)`, therefore it is added. Finally, the reaching input arguments of `(TaskB, 1)`, which are `(TaskC, 0)` and `(Finish, 0)` are added. Since `(TaskC, 0)` is not written back to URF and `(Finish, 0)` is deallocated in the task function, the buffer field ends at `TaskC` and `Finish`. Figure 6.3(b) lists the elements of the buffer web for the example control program.

In the remainder of the chapter, in every phase of code transformations, each dynamic buffer and structure accessed in the control program will be referred with the buffer/structure webs, rather than the individual Sarek variables.

6.3 Parameter Deaggregation

As it is discussed in Section 5.1, parameter deaggregation recursively replaces pointers to structures by fields, until all task parameters are of primitive types. In preparation for parameter deaggregation, for each structure web, a unique Sarek variable is created for each field of the corresponding structure. These unique variables represent a specific field of a specific structure and they are used as input/output arguments when this structure is deaggregated. This ensures correct data flow in the control program after the deaggregation.

Using the unique variables of the structure webs, each task argument of each structure web is deaggregated to individual fields of the corresponding structure, both in the task call and the task function it appears in. The analysis results of the task functions, such as definition, use, allocation and deallocation of the structure fields, are used to determine if a field of a structure will be made input and/or output argument(s) of a task.

If a task argument in a structure web is allocated in a task function, all the fields of the structure are made output arguments of this task. Similarly, if a task argument in a structure web is deallocated in a task function, all the fields of the structure are elected to be input arguments of this task. This is to ensure the synchronization of the tasks via task argument dependences, such that no other task accesses a field of the structure before its allocation and no other task accesses a field of a structure after its deallocation.

In addition, if a task argument in a structure web is not allocated or deallocated in the task it appears in, the fields of the structure are made input or output arguments of the task, according to the definition and use state of each individual field. A field of a structure web is made an input argument if the field is used (before being written) or it is made an output argument if the field is defined in the task.

The conditions in which a structure is declared to be allocated or deallocated and a structure field is declared to be used and/or defined are discussed in Section 7.2.

Apart from the basics of deaggregation process discussed above, two special cases are handled in parameter deaggregation. First, since structure-pointers may be fields of structures, parameter deaggregation processes all the accessible structures from a structure-pointer (which an input argument of a task). In other words, not only scalar and pointer fields, contained in a memory structure `str`, referred to as *direct fields* of `str`, are deaggregated, but also fields that can be accessed through structure-pointer fields of `str`, which are referred to as *indirect fields* of `str`, are also deaggregated. In order to achieve this, indirect fields of a structure are made input/output arguments of a task, in the same way that the direct fields are made input/output arguments, i.e. depending whether the field is used/defined in the task and by inserting a unique Sarek variable for each field.

However, when a structure is allocated or deallocated, in order to correctly reflect the data flow in the control program after the deaggregation, direct and indirect fields of structures are treated differently. In fact, only direct fields of the allocated/deallocated structures are made output/input arguments of tasks. Because a structure field exists in

```
for each task T
  for each input argument in_arg of T
    if in_arg is an element of a structure web str_web
      if in_arg is deallocated in T
        for each field fld of str_web
          if fld is not direct field of str_web
            if structure of fld is deallocated in T
              insert unique variable for fld to input arguments of T
          else
            insert unique variable for fld to input arguments of T
      else
        if in_arg is not used in T
          remove in_arg from the input arguments of T
        for each field fld of the str_web
          if fld is used in T
            insert unique variable of fld to input arguments of T
          else
            if a region of fld is used or defined in T
              insert unique variable of fld to input arguments of T
          if fld is defined in T
            insert unique variable of fld to output arguments of T
          else
            if a region of fld is used or defined in T
              insert unique variable of fld to output arguments of T
  for each output argument out_arg of T
    if out_arg is an element of a structure web str_web
      if out_arg is allocated in T
        for each field fld of str_web
          if fld is not direct field of str_web
            if structure of fld is allocated in T
              insert unique variable for fld to output arguments of T
          else
            insert unique variable of fld to output arguments of T
      else
        if out_arg is not defined in T
          remove out_arg from the output arguments of T
```

Figure 6.4: Algorithm for parameter deaggregation

the memory only after the allocation of the structure that it is stored in, indirect fields are made output arguments only if the structure, that the indirect fields are stored in, is allocated. Similarly, only direct fields of a structure are made input arguments when the structure is deallocated.

The second special case handled by the parameter deaggregation is the usage of buffer-pointers as fields of structures. If a buffer section accessed via a buffer-pointer `ptr`, which is a field of a structure `str`, is defined or used in a task `T`, `ptr` is made both an input and an output arguments to `T`, in order to serialize tasks that access the same buffer, via synchronization false dependences, as discussed in Section 4.2.1.

The algorithm for parameter deaggregation is depicted in Figure 6.4. Figure 6.5 and Figure 6.6 illustrates the algorithm with an example. In the input control program shown in Figure 6.5(a), for simplicity, the results of the task function analyses are shown as comments to task calls inside the control program. Figure 6.5(b) depicts the definitions of the structures accessed in the task functions. Figure 6.5(c) shows the structure webs that exist in the sample control program. Since `str1` and `str2` represent every element of the `structure web 0` and `structure web 1` respectively, through the remainder of the example, structure webs will be referred as `str1` and `str2` for simplicity.

When the sample control program (Figure 6.5(a)) is given to parameter deaggregation as input, together with the corresponding structure definitions (Figure 6.5(b)) and structure webs (Figure 6.5(c)), first, a unique Sarek variable is created for each direct and indirect field of each structure web as shown in Figure 6.6(a). Then, these unique Sarek variables are inserted to input and output argument lists of tasks according to the parameter deaggregation algorithm. First, since `str1` and `str2` are allocated in `Init`, every direct field of `str1` and `str2` are made an output argument of `Init`. In addition, because `small` field of `str2`, which is of type pointer-to-structure, is also allocated in the task function of `Init`, the single direct field of `str2->small`, which is `str2->small->b` with corresponding unique Sarek variable `str2_small_b` is also made an output argument of `Init`. On the contrary, since `small` field of `str1` is not allocated in `Init`, the unique Sarek variable `str1_small_b`, corresponding to `b` direct field of `str1->small`, does not appear in the output arguments of `Init`. Further, the input and output arguments of `TaskA`, `TaskB` and `TaskC` are modified according to the use and definition of the each field of `str1` and `str2`. As the `small` field of `str1` is allocated in `TaskA`, `str1_small` and `str1_small_b` appear as the output arguments of `TaskA`. In addition, sections of the buffer field `buf` of `str1` and `str2` are used and defined in `TaskA`, `TaskB` and `TaskC`. Thus, corresponding unique Sarek variables `str1_buf` and `str2_buf` appear as both input and output arguments of these tasks. It is important to note that `str2_a` in `TaskA` is neither

```
// Allocates str1 and str2,
// which are of type (struct big_struct *)
// Allocates str2->small
// which is of type (struct small_struct *)
Init(out str1, out str2);

// Uses str1->a
// Defines str1->a, str2->small->b, str1->buf[0:20]
// Allocates str1->small
TaskA(in str1, in str2, out str1, out str2);

// Uses str2->small->b, Defines str2->buf[0:10]
TaskB(in str2, out str2);

// Uses str1->a, Uses str1->buf[0:20]
TaskC(in str1, out str1);

// DeAllocates str1->small and str2->small
// DeAllocates str1 and str2
Finish(in str1, in str2);
```

(a) Example control program.

```
struct big_struct
{
  int a;
  int (*buf)[21];
  struct small_struct *small;
}

struct small_struct
{
  int b;
}
```

(b) Structure definitions inside the task functions.

```
Structure Web 0 = { (Init, 0),
                  (TaskA, 0), (TaskA, 2),
                  (TaskC, 0), (TaskC, 1),
                  (Finish, 0) }

Structure Web 1 = { (Init, 1),
                  (TaskA, 1), (TaskA, 3),
                  (TaskB, 0), (TaskB, 1),
                  (Finish, 1) }
```

(c) Structure webs for the sample control program.

Figure 6.5: Example control program, structure definitions and the corresponding structure webs.

input nor output arguments, because its corresponding field `str2->a` is not used and is not defined inside the task function. Similarly, none of the fields of `str1` in `TaskB` and none of the fields of `str2` in `TaskC` are arguments of their respective tasks. In addition, since `str1->small`, `str2->small`, `st1` and `str2` are deallocated in `Finish` all the fields of `str1` and `str2` are made input arguments. The resulting output control program after parameter deaggregation is shown in Figure 6.6(b).

6.4 Section Data-Flow Analysis

We use a data flow solver to propagate section definitions and uses in the CFG of the control program, in order to find shared memory dependences.

A *reaching section definition (RSD)* is a definition of a buffer section that reaches a use of an overlapping section. Similarly, a *reaching section use (RSU)* is a use of buffer section that reaches a definition of an overlapping section.

We describe the data flow analyses that are used to determine the reaching section definitions and the reaching section uses for each buffer web in a control program. These are later used to compute the flow, output and anti dependences among task calls in a control program.

The RSD analysis is a forward-any path data flow analysis that determines for each section use of a buffer (represented by a buffer web), all the reaching definitions of the same section. The sets `RSin(i)` and `RSout(i)` represents the reaching section definitions at the beginning and exit of each basic block. The set `RSgen(i)` is the set of section definitions for all the buffer webs, generated by basic block `i`. The set `RSkill(i)` represents the set of section definitions killed by basic block `i`. This includes all the section definitions for all the buffer webs in basic block `i`. It also includes the allocated section (if any) for each buffer web in basic block `i`. The data flow equations for the reaching section definition analysis are:

```
str1->a : str1_a
str1->buf : str1_buf
str1->small : str1_small
str1->small->b : str1_small_b

str2->a : str2_a
str2->buf : str2_buf
str2->small : str2_small
str2->small->b : str2_small_b
```

(a) Unique Sarek variables for example structure webs.

```
// Allocates str1 and str2,
// which are of type (struct big_struct *)
// Allocates str2->small
// which are of type (struct small_struct *)
Init(out str1, out str1_a, out str1_buf,
     out str1_small,
     out str2, out str2_a, out str2_buf,
     out str2_small, out str2_small_b);

// Uses str1->a
// Defines str1->a, str2->small->b, str1->buf[0:20]
// Allocates str1->small
TaskA(in str1_a, in str1_buf,
      out str1_a, out str1_buf, out str1_small,
      out str1_small_b, out str2_small_b);

// Uses str2->small->b, Defines str2->buf[0:10]
TaskB(in str2_small_b, in str2_buf,
      out str2_buf);

// Uses str1->a, Uses str1->buf[0:20]
TaskC(in str1_a, in str1_buf, out str1_buf);

// DeAllocates str1->small and str2->small
// DeAllocates str1 and str2
Finish(in str1, in str1_a, in str1_buf,
       in str1_small, in str1_small_b,
       in str2, in str2_a, in str2_buf,
       in str2_small, in str2_small_b);
```

(b) The output control program of parameter deaggregation.

Figure 6.6: Parameter deaggregation example.

```
RSin(i) = $\bigcup_{s \in Pred(s)}$ RSout(s)
RSout(i) = RSgen(i) $\bigcup$ [ RSin(i) - RSkill(i) ]
where initially, RSin(i) = $\phi$ $and$ RSout(i) = RSgen(i)
```

The union "$\bigcup$" operator, used in the data flow equations, is the set union operator. The "$-$" operator is defined as follow:

```
Set$_1$ - Set$_2$ =
for every element elem$_1$ of Set$_1$
    for every element elem$_2$ of Set$_2$
        if elem$_1$ and elem$_2$ are from the same buffer web
            Diff(elem$_1$, elem$_2$)
```

where the `Diff` operator is defined as

$$Diff([a:b],[c:d]) = \begin{cases} [a:c-1] & \text{if } a < c <= b <= d; \\ \phi & \text{if } c <= a <= b <= d; \\ [d+1:b] & \text{if } c <= a <= d < b; \\ [a:c-1] \bigcup [d+1:b] & \text{if } a < c <= d < b; \\ [a:b] & \text{if } a <= b < c <= d; \\ [a:b] & \text{if } c <= d < a <= b. \end{cases}$$

The RSUs are computed using the same data flow equations and $\bigcup$ and `-` operators, as RSDs above. However, for RSU analysis, RSgen(i) includes, for all the buffer webs, all the used but not defined sections of the basic block `i`, while the RSkill(i) set, includes the allocated and defined sections, if any, in the basic block `i`.

Using the RSDs and RSUs, three types of data-flow relationships among task calls, caused by accesses to buffers, are computed for each buffer web: flow, output and anti dependences. These dependences are represented with directed edges from the source task calls of the dependences to the sink task calls of the dependences, in several graphs that will be discussed in Section 6.5. These edges are called *flow*, *output* and *anti* edges and they are determined according to the following rules:

Flow Edges: links the task call $\texttt{T}_1$ of a section definition $\texttt{def}_1$ to the task call $\texttt{T}_2$ of an overlapping section use $\texttt{use}_2$ (from the same buffer web) that $\texttt{def}_1$ reaches to in the reaching section definitions.

Output Edges: links the task call $\texttt{T}_1$ of a section definition $\texttt{def}_1$ to the task call $\texttt{T}_2$ of an overlapping section definition $\texttt{def}_2$ (from the same buffer web) that $\texttt{def}_1$ reaches to in the reaching section definitions.

Anti Edges: links the task call $\texttt{T}_1$ of a section use $\texttt{use}_1$ to the task call $\texttt{T}_2$ of an overlapping section definition $\texttt{def}_2$ (of the same buffer web) that $\texttt{use}_1$ reaches to in the reaching section uses.

In the control program depicted with the Figure-6.7(a), there is a single buffer web, shown in Figure-6.7(b).

Since in the example control program of Figure 6.7(a), each task is called once, task calls are referred by the name of the task called. However, generally, task calls, which are the computation units of section data flow analysis, do not form a one-to-one mapping with the tasks; therefore, they should be referenced with the basic block that the task call is in.

Furthermore, in the remainder of this chapter, for a task call, the list of reaching section definitions are given on the right hand side of "=" operator. A reaching section is represented with a section inside brackets followed by the task call that the section is generated in.

The reaching section definitions are listed in Figure 6.8(a). In the figure, it is seen that no section definition is reaching `Fill_Buffer`. The reason is the allocation of the buffer web in task `Init`, which is killing any reaching definition from the previous iteration of the loop. Similarly, the `Update_Result` task is killing the [0:20] section of the buffer and consequently, only [21:40] section is reaching from the `Fill_Buffer` task to `Output_Results` task.

```
while(cont)
{
  //Allocates buffer
  Init(out buf);

  //Defines buf[0:40]
  cont = Fill_Buffer(in buf);

  //Uses buf[0:20]
  Process_Buffer(in buf);

  while(correct)
  {
    //Uses buf[21:40] and Defines buf[0:20]
    correct = Update_Results(in buf);

    //Uses buf[0:40] and Defines buf[0:40]
    Output_Results(in buf);
  }

  //DeAllocates buf
  Finish(in buf);
}
```

(a) Sample control program.

```
Buffer Web 0 = { (Init, 0),
                 (Fill_Buffer, 0),
                 (Process_Buffer, 0),
                 (Update_Results, 0),
                 (Output_Results, 0),
                 (Finish, 0) }
```

(b) Buffer webs.

Figure 6.7: Example input control program and the corresponding input buffer webs for the section data flow solver.

```
Init = { [0:40]:Fill_Buffer,
         [0:40]:Output_Results }

Fill_Buffer = { }

Process_Buffer = { [0:40]:Fill_Buffer }

Update_Results = { [0:40]:Fill_Buffer,
                   [0:40]:Output_Results }

Output_Results = { [0:20]:Update_Results,
                   [21:40]:Fill_Buffer,
                   [21:40]:Output_Results }

Finish = { [0:40]:Fill_Buffer,
           [0:40]:Output_Results }
```

(a) Reaching section definitions for `Buffer Web 0` of the example control program.

```
Init = { [0:20]:Process_Buffer }

Fill_Buffer = { }

Process_Buffer = { }

Update_Results = { [0:20]:Process_Buffer }

Output_Results = { [21:40]:Update_Results }

Finish = { [0:20]:Process_Buffer }
```

(b) Reaching section uses for `Buffer Web 0` of the example control program.

Figure 6.8: Section data flow example.

```
Fill_Buffer -> Process_Buffer
Fill_Buffer -> Update_Results
Output_Results -> Update_Results
Update_Results -> Output_Results
Fill_Buffer -> Output_Results
Output_Results -> Output_Results
```

(a) Flow edges.

```
Fill_Buffer -> Update_Results
Output_Results -> Update_Results
Update_Results -> Output_Results
Fill_Buffer -> Output_Results
Output_Results -> Output_Results
```

(b) Output edges.

```
Process_Buffer -> Update_Results
Update_Results -> Output_Results
```

(c) Anti edges.

Figure 6.9: Flow, output and anti edges.

Figure 6.8(b) lists the reaching uses for the example control program of Figure 6.7(a). Since uses can be killed by allocations, `Init` task kills any reaching section use and no section use reaches `Fill_Buffer` and `Process_Buffer`. Further, the section use in `Process_Buffer` reaches `Finish` and `Update_Results`, but since `Update_Results` defines the exact same section, it is killed in `Update_Results` and does not reach `Output_Results`. Similarly, the section use in `Update_Results` reaches `Output_Results`, where it is killed. On the other hand, the section use in `Output_Results` is killed by the section definition inside the same task and, thus, it does not reach any task.

Figure 6.9 illustrates flow, output and anti edges for the example control program shown in Figure 6.7(a), starting with the reaching section definition and uses shown in Figure 6.8.

6.5 Dependence Graphs

The transformations we describe in the remainder of this chapter require use of several dependence graphs. These graphs are described in this section.

A *dependence graph* is a directed multi-graph $G = (V, E)$ whose vertices V are the task calls of the control program and the edges E are the dependence relations among task calls. In the dependence graph, there is an edge from a task call $\mathtt{T}_1$ to another task call $\mathtt{T}_2$ if and only if $\mathtt{T}_2$ is data-dependent on $\mathtt{T}_1$.

If a dependence graph includes dependence relations caused by the accesses to a single buffer (represented by a buffer web), the graph is called a *single-buffer dependence graph.* However, if dependences caused by accesses to URF and also to all the buffers in the control program are represented in a single dependence graph, this graph is called *multi-variable dependence graph.* In the multi-variable dependence graph of a control program, there is an edge from a task call $\mathtt{T}_1$ to another task call $\mathtt{T}_2$, if there is a data-dependence from $\mathtt{T}_1$ to $\mathtt{T}_2$ caused by any buffer in the memory or any Sarek variable in the control program. Similarly, a dependence is said to exist from $\mathtt{T}_1$ to $\mathtt{T}_2$, if there is an edge from $\mathtt{T}_1$ to $\mathtt{T}_2$ in the multi-variable dependence graph.

In a dependence graph `G`, two vertices v_i and v_j are said to be *connected*, if there is a path from v_i to v_j in `G`. The connection relationship partitions the graph into subgraphs $V_1,\ V_2,\ ...,\ V_k$. These subgraphs are called the *connected components (CC)* of the dependence graph. The non-overlapping largest connected components are called *maximally connected components (MCC).* A pair of vertices (i.e. task calls) are connected if and only if they belong to the same MCC of the data dependence graph.

A task call $\mathtt{T}_1$ is said to be *independent* of another task call $\mathtt{T}_2$, according to a dependence graph $\mathtt{G}$, if there is no path from $\mathtt{T}_2$ to $\mathtt{T}_1$ in $\mathtt{G}$. Conversely, a task call $\mathtt{T}_1$ is said to be *dependent* on another task call $\mathtt{T}_2$, according to a dependence graph $\mathtt{G}$, if there is at least one path from $\mathtt{T}_2$ to $\mathtt{T}_1$ in $\mathtt{G}$.

Dependence graphs $G = (V, E)$ are categorized into three types in terms of the data dependence it represents: *flow dependence graphs (FDGs)*, *output dependence graphs (ODGs)* and *anti dependence graphs (ADGs)*.

The FDG is a graph $G_f = (V, E_f)$ whose vertices are the task calls of the control program and the edges E_f are the flow edges. The ODG is a graph $G_o = (V, E_o)$ whose vertices are the task calls of the control program and the edges E_o are the output edges. Finally, the ADG is a graph $G_a = (V, E_a)$ whose vertices are the task calls of the control program and the edges E_a are the anti edges. In other words, the graphs respectively represent the flow, output and anti dependences among task calls caused by accesses to buffers in the memory and they consist of flow, output and anti edges.

FDGs are particularly important in the sense that they represent unbreakable constraints for the run-time execution of tasks. Two independent task calls, according to the FDG, may execute in parallel at run-time. On the other hand, two dependent tasks will definitely be serialized during execution.

In order to perform the buffer privatization, buffer replication and buffer renaming code transformations, an FDG, an ODG and an ADG are constructed for each buffer web. In addition, a multi-variable flow-dependence graph is built to represent all the flow dependences among all the task calls, caused by accesses to both the shared-memory and URF registers. For this purpose, first, pointer type Sarek variables are excluded from the def-use chains of task arguments (as discussed in Section 6.1) and flow dependences are computed for the remaining scalar type task arguments, similar to the way they are computed for buffer webs, as discussed in Section 6.4. Then, these dependences and the FDGs of all the buffer webs are merged to form the multi-variable dependence graph.

Figure 6.10 depicts the FDG, ODG and ADG of `buf1` in the example control program shown in Figure-6.7(a) with the flow, output and anti edges listed in Figure 6.9.

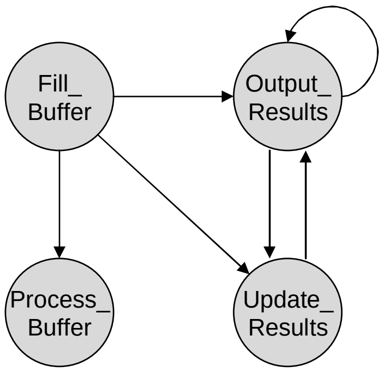
6.6 Buffer Privatization

Buffer Privatization aims to break false dependences caused by the accesses to the same buffer, as it is described in Section 5.2.2. It does so by creating private copies of the buffers for certain tasks, allowing these tasks to execute in parallel with each other as well with the remaining tasks. In this section, the details of buffer privatization are presented.

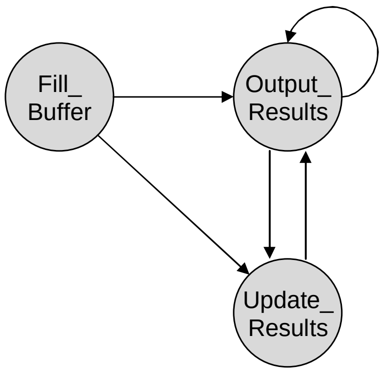
Buffer privatization is implemented in six steps, using the buffer webs, FDG of each buffer web and the multi-variable FDG of the control program.

First, maximally connected components of the FDG of each buffer web are determined. Each of these components represents the set of task calls in which every task call is dependent to only the task calls from the same component. These connected components are the candidates for buffer privatization. Second, each connected component is evaluated to determine if it is eligible for privatization. Third, the head and tail elements of each connected component are found. Fourth, by using the multi-variable FDG of the control program, the eligible connected components are merged, in order to perform the privatization more efficiently. Fifth, the merged eligible connected components are evaluated for efficiency conditions. Finally the eligible components are privatized in the control program using the helper tasks and the task functions of these helper tasks are created according to the type of the privatized buffer.

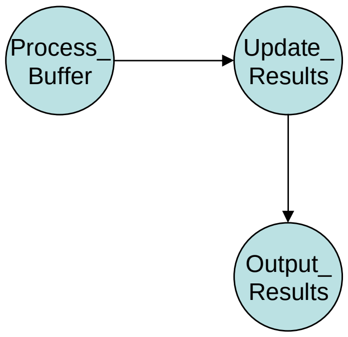
The following sections examine each step of the buffer privatization in detail. These steps are illustrated with the help of a control program shown in Figure 6.11. Throughout any analysis required, the section definition, section use, allocation and deallocation information is taken from the results of the analyses of the task functions. However, for



(a) Flow dependence graph (FDG).



(b) Output dependence graph (ODG).



(c) Anti dependence graph (ADG).

Figure 6.10: Dependence graphs.

ease of presentation, these are shown as comments in the control program.

For the control program in Figure 6.11, the Sarek variables of `buf1` and `buf2` carry the same value of the pointer throughout the execution of the control program. Therefore, through the remainder of the section, for simplicity, we will refer to the buffer webs as `buf1` and `buf2`, even though Sarek variables do not necessarily represent buffer webs in general.

After the flow edges are created for each buffer web, the resulting multi-variable FDG of the control program (i.e. for buffer webs `buf1` and `buf2` and the Sarek variable `count`) is obtained as depicted in Figure 6.12.

In the following sections, buffer privatization will be illustrated for `buf1` of the example program. Therefore, FDG of `buf1`, shown in the Figure 6.13, will be used in the steps of the buffer privatization.

6.6.1 Finding Maximally Connected Components

By the definition of an MCC (mentioned in Section 6.5), two vertices (i.e. task calls) are connected (i.e. dependent) to each other if and only if they are in the same MCC. Therefore, in an FDG of a buffer web `buf`, MCCs represent the collection of task calls, in which each task call is dependent only on task calls of the same MCC, in terms of accesses to `buf`. Thus, tasks in different MCCs of the FDG of `buf` do not access (read/write) the same data in `buf`. Consequently, if task calls in an MCC access (read and write) a different buffer `buf_new` in memory, the tasks in other connected components will be not affected, as they will still access the correct data in `buf`. In fact, task calls in each MCC can safely access different buffers in the memory without affecting the correctness of the overall program. Thus, MCCs of FDGs are considered as the privatization candidates, for which a private copy of the buffer (that MCC belongs to) can be reserved.

Furthermore, MCCs of FDGs are minimal self-dependent units of privatization, in the sense that, removing even a single vertex (i.e. task call) from an MCC, would break the

```
// Allocates buf1 and buf2 of type int(*)[21]
// count is of type int
Init(out buf1, out buf2, out count);

cont1 = 0x1;
while(cont1)
{
  // Defines buf1[0:10]
  cont1 = Task1(in buf1, out buf1);

  cont2 = 0x1;
  while(cont2)
  {
    // Uses buf1[0:10]
    // Defines buf2[0:20]
    cont2 = Task2(in buf1, in buf2, out buf1, out buf2);

    // Uses buf1[11:20]
    // Defines buf1[11:20]
    // Uses buf2[0:20]
    Task3(in buf1, in buf2, out buf1, out buf2);

    // Uses buf1[11:20]
    // Defines buf1[11:20]
    Task4(in buf1, out buf1);

    // Uses buf1[11:20]
    // Defines buf1[11:20]
    Task5(in buf1, out buf1);

    // Defines buf1[0:10]
    Task6(in buf1, out buf1);

    // Uses buf1[0:10]
    // Defines buf2[0:20]
    Task7(in buf1, in buf2, out buf1, out buf2);

    // Defines buf1[0:10]
    // Uses buf2[0:20]
    Task8(in buf1, in buf2, out buf1, out buf2);

    // Uses buf1[0:10]
    Task9in buf1, out buf1);
  }

  cont3 = 0x1;
  while(cont3)
  {
    // Defines buf1[0:20]
    cont3 = Task10(in buf1, out buf1, in count);

    // Uses buf1[0:20]
    // Defines buf2[0:20]
    Task11(in buf1, in buf2, out buf1, out buf2);

    // Uses buf2[0:20]
    Task12(in buf2, out buf2, out count);
  }
}

// Defines buf1[0:20]
Task13(in buf1, out buf1);

// Uses buf1[0:20]
Task14(in buf1, out buf1);

// Deallocates buf1 and buf2
Finish(in buf1, in buf2);
```

Figure 6.11: Example control program for buffer privatization.

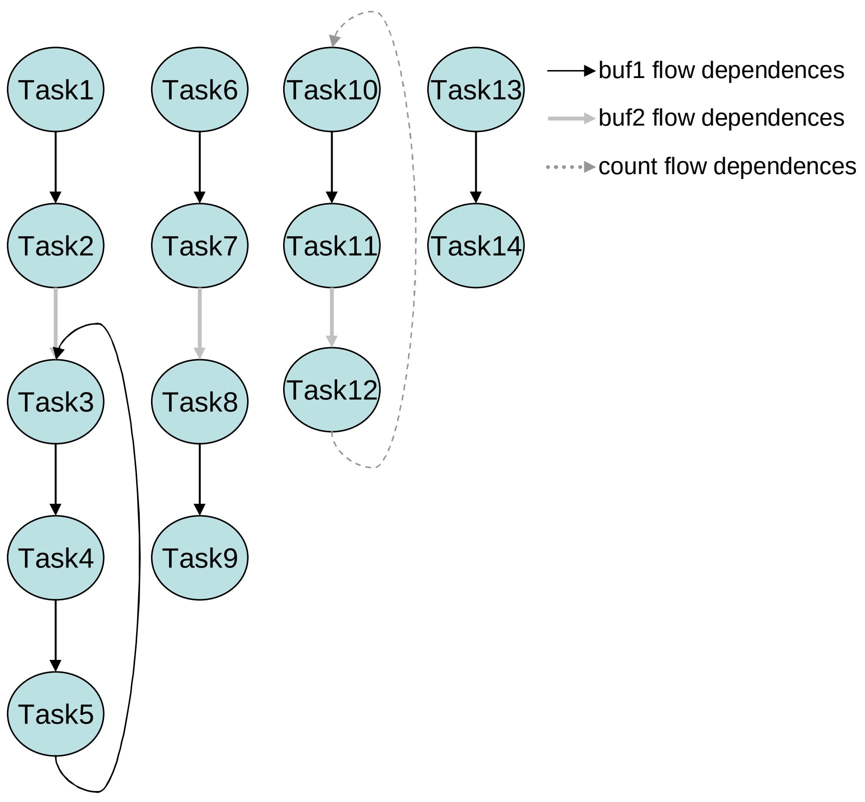


Figure 6.12: Multi-variable FDG for the example control program.

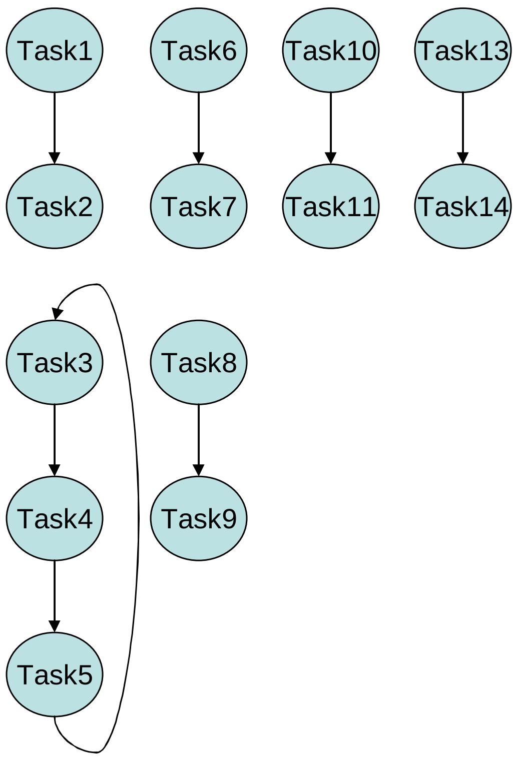


Figure 6.13: FDG for `buf1`.

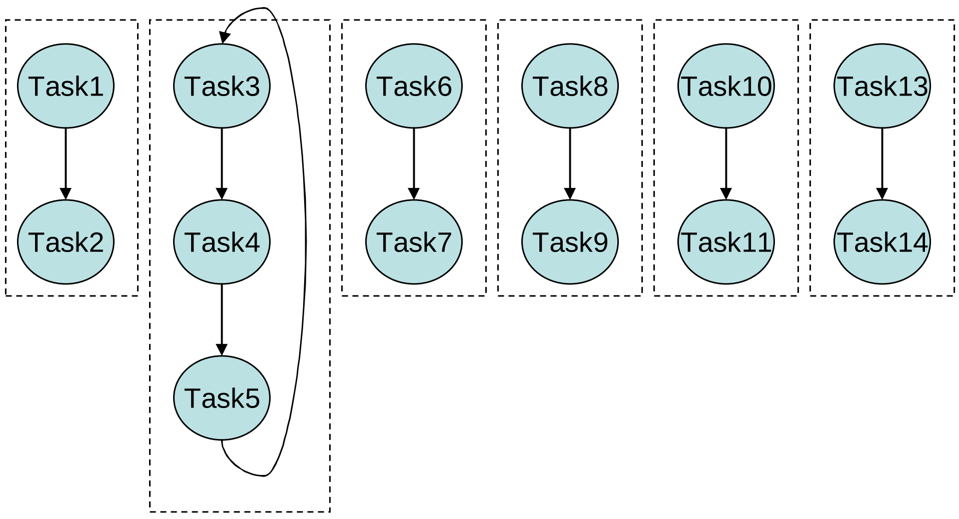


Figure 6.14: MCCs for the running example.

flow dependences and result to incorrect execution when the MCCs are reserved private buffers. However, merging multiple MCCs is always correct, because no flow dependences are broken.

Figure 6.14 depicts the connected components of the data-flow graph of `buf1`, for the example control program in Figure 6.11.

6.6.2 Heads and Tails of Maximally Connected Components

A *head element* is an element of a MCC that is not dependent on any task call of the MCC, whereas a *tail element* is an element of a MCC that no other task call of the MCC depend on. Effectively, the head and tail elements are the entry and exit points of a connected component.

An element `n` is a head element if `n` is not a sink node in any flow edge of the MCC. Similarly, an element `n` is a tail element if `n` is not a source node in any flow edge of the MCC. In an MCC, there may be more than one head or tail elements or there may be none.

The head and tail elements of each MCC are computed in order to be used in eval-

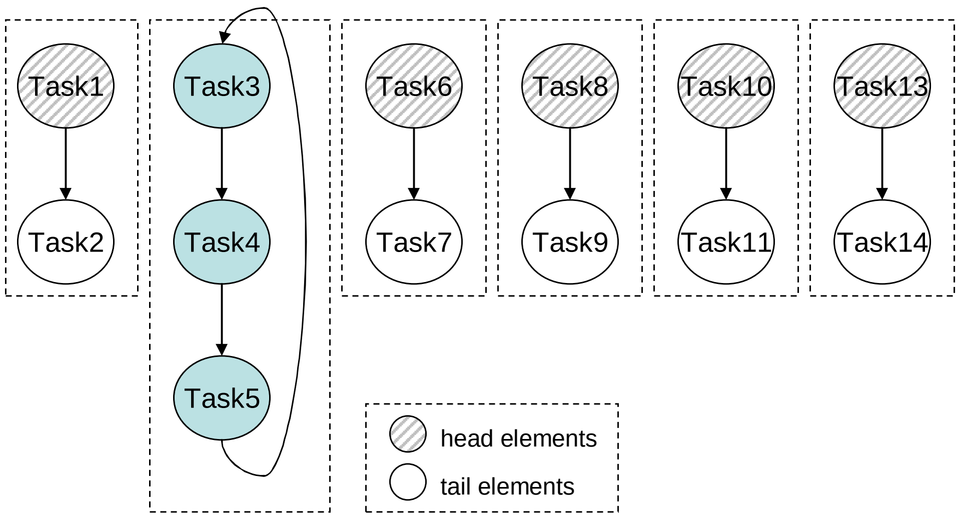


Figure 6.15: Head and tail elements for the running example.

uating privatization conditions and determining the boundaries of privatization in the control program as will be discussed in Section 6.6.6.

The head and tail elements for the connected components shown in Figure 6.14 are depicted in Figure 6.15. In the figure, no head and tail elements are found in the MCC consisting of `Task3`, `Task4` and `Task5` because every task call is a sink node and also a source node in at least one flow-edge of the MCC. On the other hand, the remaining MCCs consist of single flow-edges. Thus, the source nodes of the flow-edges are selected as head elements, whereas the sink nodes are selected as tail elements.

6.6.3 Finding Eligible Connected Components

The MCCs of a buffer `buf`, is the units of privatization that can access a private copy of `buf`. As discussed in Section 5.2.2, privatization can happen in the main body of the control program or inside the body of a loop, depending where the private buffer is allocated and destroyed. In the first case, a private copy will be created once, for the task calls (of the privatized MCC of `buf`), so that these tasks can safely execute in parallel with the remaining task calls that are accessing `buf`. However, in the latter case, a separate

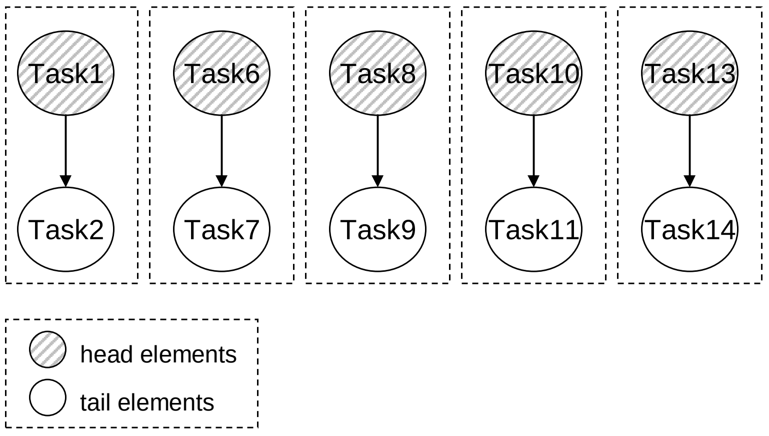


Figure 6.16: Eligible MCCs.

private copy of `buf` will be created in every iteration of the loop, so that task calls from different iterations can safely execute in parallel. For this approach to generate correct programs, the task calls in one iteration should not use data produced in the previous iterations. In other words, there should not exist loop-carried data dependences in terms of accesses to `buf`.

In order to find the tasks that are eligible for buffer privatization, each MCC of each buffer web is checked for loop-carried data dependences. No loop-carried data dependence exists in an MCC, for every flow-edge of the MCC, if the source node is an ancestor of the sink node or the sink node is a descendent of the source node, in the CFG of the control program, i.e. the source node of the dependence executes earlier than the sink node of the dependence. The MCCs that satisfy this condition are called *eligible MCCs*.

When the eligibility condition of the buffer privatization is applied to the privatization candidates shown in Figure 6.15, the MCC consisting of `Task3`, `Task4` and `Task5` is eliminated due to the loop-carried data-dependence from `Task5` to `Task3`, which would be broken by privatization. Figure 6.16 list the MCCs eligible for buffer privatization.

It is important to note that if a MCC does not contain any loop-carried dependence, it is guaranteed to have at least one head and one tail element. This is due to the fact that, if there exist a tail element `t` and a head element `h` of an MCC, then there exists a path starting from `h` and ending at `t`. A tail or a head element do not exist, only if there are circular paths in the graph. Since circular paths requires loop-carried dependences to exist, if an MCC does not contain any loop-carried dependence, at least one head and one tail element exist in this MCC.

6.6.4 Merging Eligible Maximally Connected Components

After the MCCs containing loop-carried dependences are eliminated, each eligible MCC can safely be assigned a separate copy of the privatized buffer. Although this approach does result in correct execution, it may not be the most efficient. This is because buffer privatization introduces a run-time overhead to the execution of the control programs which should be avoided, if privatization does not lead to additional parallel execution. Therefore, privatization is applied to groups of MCCs, rather than for every single MCC, as long as performance is not affected. In this step, for each buffer web, the eligible MCCs that can share a single buffer with no loss in parallelism are merged to form a single privatization unit.

First, by the definition of an MCC, task calls in two MCCs `m1` and `m2` that belong to `buf` are not dependent on each other in terms of accesses to `buf`. However, they can be dependent, in terms of accesses to other buffers in the memory or scalars in the URF, in which case they can not execute in parallel. If these dependences among task calls of `m1` and `m2`, caused by accesses to other buffers and scalars in URF, are true dependences, they can not be broken by the hardware or by any other code transformation. In this case, since task calls in `m1` and `m2` can never execute in parallel, assigning a separate private copy to `m1` and `m2` has no benefit over assigning a single private copy to both `m1` and `m2`. Consequently, two MCCs that are dependent on each other in terms of accesses to any buffer in the memory or scalar in the URF are merged to form a single privatization candidate.

In order to efficiently find out if task calls in two MCCs `m1` and `m2`, that belong to a buffer `buf`, are dependent on each other, the dependences among head and tail elements of `m1` and `m2` are checked, in the multi-variable flow dependence graph of `buf`. If all the head elements of `m2` are dependent on at least one tail element of `m1` or, similarly, all the head elements of `m1` are dependent on at least one tail element of `m2`, none of the task calls of `m2` can execute in parallel with the task calls of `m1`. For all the buffer webs in the control program, the MCCs that satisfy this condition are merged to form a single privatization candidate, thus to avoid any privatization overhead. It is important to note that in this merging step, every MCC will have at least one head element and one tail element, because in the previous step the MCCs with loop-carried dependences, that may not have head or tail elements, have already been eliminated.

Furthermore, the fact that task calls in two MCCs `m1` and `m2` of a buffer `buf` do not have true dependences with each other in terms of accesses to `buf` (by the definition of MCC), does not require that they have false dependences with each other. In case no false dependence (output or anti) exists among task calls of `m1` and `m2`, these task calls can execute in parallel even if they are accessing the same buffer. Consequently, separate privatization for `m1` and `m2` will not result in any performance gain. In order to avoid unnecessary privatization overheads, when there are no false dependences among task calls of two MCCs, these two MCCs are merged to form a single privatization candidate.

The false dependences among task calls of different MCCs are checked by the accessed regions of each MCC. If the total region accessed (written and read) by task calls of an MCC does not overlap with the total region accessed by the task calls of another MCC, false dependences can not exist among task calls of these two MCCs. Therefore, for each buffer web and for each MCC, total accessed regions are found by merging the defined and used sections of the buffer, that MCC belong to, for each task call in the MCC. Next, for each buffer web, MCCs that have non-overlapping total access regions are merged to form a single privatization candidate.

After the merge operation, as the merged connected components do not necessarily

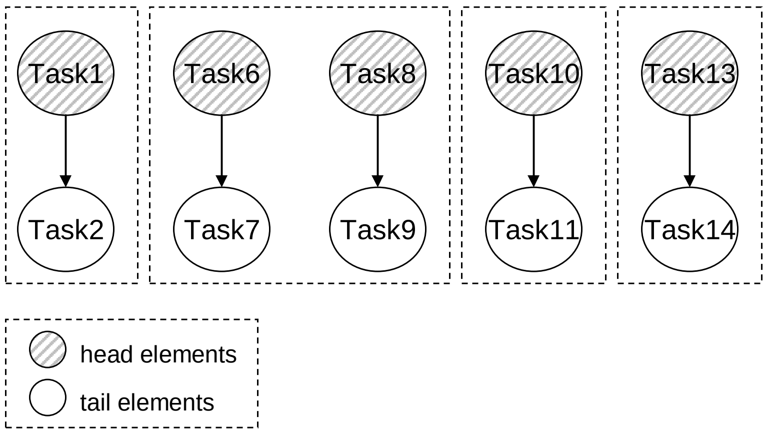


Figure 6.17: Merged components for the running example.

include connected vertices, each merged connected component is referred as only a *merged component (MC)*.

The MCs depicted in Figure 6.17 are obtained for `buf1`, after the eligible MCCs of `buf1` shown in Figure 6.16 are merged. It is seen that the MCC consisting of `Task6` and `Task7` is merged with the MCC of consisting of `Task8` and `Task9`. This is because of the flow dependence from `Task7`, which is the tail element of its MCC, to `Task8`, which is the head element of its MCC. This flow dependence is caused by the accesses to `buf2` and can be seen in the multi-variable FDG of the control program shown in Figure 6.12.

6.6.5 Finding Efficient Eligible Components

When an MC `m` is the only MC that is included in a loop body `l` (including loops nested in `l`) of the CFG of the control program, the only parallelism that can be obtained by privatizing `m` is the loop-level parallelism. Thus, if privatizing `m` does not result in loop-level parallelism, no gains are realized through privatization. In order to avoid unnecessary overhead, each MC of each buffer is checked to ensure that privatization will result in loop-level parallelism.

A *cycle* is said to exist among task calls of an MC `m` that belongs to `buf`, if all the head elements of `m` are dependent on the tail elements of `m`, in terms of accesses to *other* buffers in the memory or to scalars in the URF. Cycles prevents the overlap of execution of the task calls of `m` inside the loop of `m`, preventing the loop-level parallelism. Consequently, if an MC of a buffer web, is the only MC in its loop (containing nested loops) and a cycle exists among its task calls, this MC is eliminated, because it is not promising neither loop-level parallelism nor body-level parallelism.

Furthermore, if no loop includes all the task calls of an MC `m`, only body-level parallelism can be obtained in the main body of the control program. In case this `m` is also the only MC in the main body, the opportunity of body-level parallelism will be lost. Thus, for each buffer web, if there exist a single MC `m` such that no loop contains all the task calls of `m`, `m` is marked as not-eligible for privatization.

After this step, the remaining MCs of each buffer web can safely and efficiently privatized; thus, they are called *efficient-eligible components (EEC)*.

Among the MCs depicted in Figure 6.17, the MC including task calls of `Task10`, `Task11` and `Task12` is discarded because of the data dependence from `Task11` to `Task10`. Even though there is no direct flow-dependence from `Task11` to `Task10`, the data dependence caused by `buf2` (shown in Figure 6.12) which is from `Task11` to `Task12` and the data dependence caused by scalar Sarek variable `count` which directs from `Task12` to `Task10` form a dependence path from `Task11` to `Task10` causing a cycle. Although this cycle is not caused by accesses to `buf1`, it eliminates loop-level speedup gain obtainable by privatizing `buf1` for the mentioned MCC task calls. Since this MCC is the only MCC that is located inside the loop `while(cont3)`, body-level parallelism can not realize for the task calls inside it, and, therefore, it is eliminated. The remaining MCs, forming EECs, and are shown in Figure 6.18.

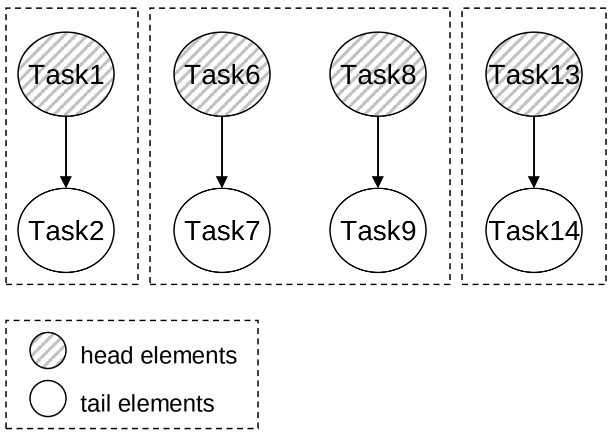


Figure 6.18: Efficient-eligible components for the running example.

6.6.6 Privatization In The Control Program

The tasks in the EECs constitute the buffer privatization targets, i.e. the tasks that will access private copies of the buffer.

The mechanics of the privatization in the control program are as follows:

1. A private buffer is created for each EEC.

2. This private buffer is passed to the target tasks instead of the original buffer by means of renaming the Sarek arguments carrying the original buffer pointer with the Sarek arguments carrying the value of the private buffer pointer.

3. Private buffer is destroyed after it is accessed by all the target tasks.

A crucial question arises at this point about where to create and destroy the private copy, referred as *the boundaries of privatization.* Valid locations for allocation and destruction of the private buffer should satisfy the following conditions:

1. Allocation and destruction should either be in the same loop or in the main body of the control program.

2. Allocation location should be earlier than all the tasks of the component in terms of the program execution.

3. Destruction location should be later than all the tasks of the component in terms of the program execution.

4. The execution distance between the allocation and destruction should be minimal in order not to increase the physical register pressure.

To satisfy the above conditions, the following scheme is used to find the allocation and destruction locations:

1. As all the privatization candidates are dependent on the head elements, head elements executes earlier than all the privatization candidates. Therefore, allocation location should be earlier than all the head elements of the EEC task calls.

2. As all the tail elements depend on the other privatization candidates, tail elements executes later than all the privatization candidates. Therefore, destruction location should be later than all the tail elements of the EEC task calls.

In order to follow the above described rules, we define the *allocation location* for a privatization candidate as the basic block location that dominates all the head elements and is inside the innermost loop body that includes all the task calls of the EEC. Similarly, we define the *destruction location* for a privatization candidate as the basic block location that post-dominates all the tail elements and that is inside the same loop body as the allocation location.

After the allocation and destruction locations are found; buffer privatization is realized with the insertion of two helper task calls.

1. `Init` task allocates the private buffer at the allocation location and has a single output argument, i.e. private buffer pointer, and no input arguments. The format of an `Init` task call is as follows:

   `Init_P_n(out buffer_p_n); where n is the unique buffer privatization id.`

2. `Finish` task deallocates the private buffer at the destruction location and has a single input argument, i.e. private buffer pointer, and no output arguments. The format of a `Finish` task call is as follows:

   `Finish_P_n(in buffer_p_n); where n is the unique buffer privatization id.`

After the described buffer privation scheme is applied to sample program in Figure 6.11 for `buf1` with the EEC in Figure 6.17, the control program in Figure 6.19 is obtained. In the control program shown in the figure, `buf1` is privatized for `Task1` and `Task2` in the outermost loop. In addition, `buf1` is privatized for `Task6`, `Task7`, `Task8` and `Task9` in the innermost loop and for `Task13` and `Task14` in the main body of the control program.

6.6.7 Creating Helper Task Functions

After the privatization in the control program is achieved, the last step of the privatization is to create the function bodies of the helper tasks, i.e. `Init_P_n` and `Finish_P_n`. In order to realize this, the type of the buffer is taken from the allocation node of the original buffer and a `malloc` operation (in `Init_P_n`) and a `free` operation (in `Finish_P_n`) are performed.

For the example buffer privatizations in Figure 6.19, the `Init` and `Finish` task bodies shown in Figure 6.20 are created.

6.7 Buffer Replication

Buffer privatization successfully removes memory output dependences. Thus, only memory anti dependences (among false dependences) remain at the end of buffer privatization. Consequently, buffer replication targets only memory anti dependences.

In general, buffer replication removes an anti dependence that flows from $\texttt{T}_1$ to $\texttt{T}_2$, caused by accesses to a memory buffer `buf`. This is done by making a copy of `buf` together

```
// Allocates buf1 and buf2 of type int(*)[21]
// count is of type int
Init(out buf1, out buf2, out count);

while(cont1)
{
  Init_P_1(out buffer_p_1);

  // Defines buffer_1[0:10]
  cont1 = Task1(in buffer_p_1, out buffer_p_1);

  while(cont2)
  {
    // Uses buffer_p_1[0:10]
    // Defines buf2[0:20]
    cont2 = Task2(in buffer_p_1, in buf2, out buffer_p_1, out buf2);

    // Uses buf1[11:20]
    // Defines buf1[11:20]
    // Uses buf2[0:20]
    Task3(in buf1, in buf2, out buf1, out buf2);

    // Uses buf1[11:20]
    // Defines buf1[11:20]
    Task4(in buf1, out buf1);

    // Uses buf1[11:20]
    // Defines buf1[11:20]
    Task5(in buf1, out buf1);

    Init_P_2(out buffer_p_2);

    // Defines buffer_p_2[0:10]
    Task6(in buffer_p_2, out buffer_p_2);

    // Uses buffer_p_2[0:10]
    // Defines buf2[0:20]
    Task7(in buffer_p_2, in buf2, out buffer_p_2, out buf2);

    // Defines buffer_p_2[0:10]
    // Uses buf2[0:20]
    Task8(in buffer_p_2, in buf2, out buffer_p_2, out buf2);

    // Uses buffer_p_2[0:10]
    Task9in buffer_p_2, out buffer_p_2);

    Finish_P_2(in buffer_p_2);
  }

  Finish_P_1(in buffer_p_1);

  // Defines buf1[0:20]
  Task10(in buf1, out buf1, in count);

  // Uses buf1[0:20]
  // Defines buf2[0:20]
  Task11(in buf1, in buf2, out buf1, out buf2);

  // Uses buf2[0:20]
  Task12(in buf2, out buf2, out count);
}

Init_P_3(out buffer_p_3);

// Defines buffer_p_3[0:20]
Task13(in buffer_p_3, out buffer_p_3);

// Uses buffer_p_3[0:20]
Task14(in buffer_p_3, out buffer_p_3);

Finish_P_3(in buffer_p_3);

// Deallocates buf1 and buf2
Finish(in buf1, in buf2);
```

Figure 6.19: Privatization in the control program.

```
int Init_P_1()
{
  int(*new_buffer)[21];

  new_buffer = malloc(sizeof(int) * 21);

  writeArg(0, new_buffer);

  return 1;
}
```

(a) The body of the `Init_P_1` task.

```
int Finish_P_1()
{
  int(*new_buffer)[21];

  new_buffer = readArg(0);

  free(new_buffer);

  return 1;
}
```

(b) The body of the `Finish_P_1` task.

Figure 6.20: The bodies of the `Init_P_1` and `Finish_P_1` tasks.

```
// Allocates buf1, buf2 and buf3 of type int(*)[21]
// Initializes count of type int
Init(out buf1, out buf2, out buf3, out count);

// Defines buf1[0:10]
TaskA(in buf1, out buf1);

while()
{

  // Uses buf1[0:20]
  TaskB(in buf1, out buf1);

  // Defines buf1[11:20]
  TaskC(in buf1, out buf1);

  // Uses buf2[0:20]
  // Defines buf2[0:20]
  TaskD(in buf2, out buf2);

  // Uses buf2[0:20]
  TaskE(in buf2, out buf2);

  // Uses buf2[0:20]
  // Defines buf2[0:20]
  TaskF(in buf2, out buf2);

  // Defines buf3[0:20]
  TaskG(in buf3, in buf3, in count);

  // Uses buf3[0:20]
  TaskH(in buf3, in buf3, out count);
}

// DeAllocates buf1, buf2 and buf3
Finish(in buf1, in buf2, in buf3);
```

Figure 6.21: Example control program for buffer replication.

with the data contained in `buf`. This copy is given to the source of the dependence ($\mathtt{T}_1$), whereas the sink task ($\mathtt{T}_2$) still processes the original buffer `buf`.

Buffer replication is realized in three steps: finding eligible replication candidates, performing replication in the control program and creating helper task functions. In the remainder of this section, each step of buffer replication will be described in detail with help of an example control program shown in Figure 6.21. In the figure, the results of task analyses are shown as comments to corresponding task calls. In the example control program, since each of the Sarek variables `buf1`, `buf2` and `buf3` represent a different buffer in memory, for simplicity, buffer webs will be referred with the Sarek variables.

For the example control program and the corresponding buffer webs, it is important to note that buffer privatization can not be applied to `buf1`, because of the loop-carried data dependence from `TaskC` to `TaskB` and, similarly to `buf2` because of the loop-carried data dependence from `TaskF` to `TaskD`. On the other hand, `buf3` can not be privatized due to the flow dependence caused by the scalar URF variable `count` which creates a cycle from `TaskH` to `TaskG`. Consequently, there exist unresolved anti dependences, caused by accesses to `buf1`, `buf2` and `buf3`, in the example control program.

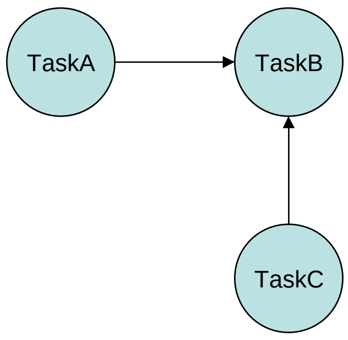
6.7.1 Finding Eligible Replication Candidates

In order to perform buffer replication, all pairs of tasks that have an anti dependence with each other are needed for each buffer in the memory. These are obtained from the ADGs of the buffer webs. Each anti dependence edge in the ADG of each buffer web is initially declared as a buffer replication candidate.

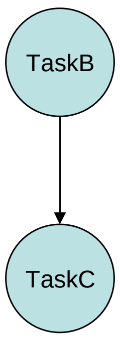
Since every false dependence can safely be resolved with memory renaming, there is no correctness condition applicable to replication candidates. In other words, when every replication candidate, i.e. every anti dependence, in a control program is resolved via replication, correct data flow in the output control program will be obtained.

However, we opt not to apply buffer replication to a buffer `buf`, if the source task $\texttt{T}_1$ of the anti dependence caused by accesses to `buf`, also writes to a section of `buf`. Such a situation will require that all other tasks that are flow dependent on $\texttt{T}_1$ in terms of accesses to `buf` be modified to use the copy. This is in general difficult to perform and may not always produce optimized code.

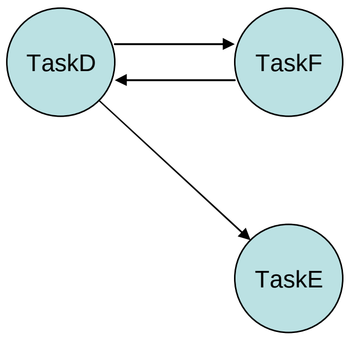
Furthermore, it may not always be possible to obtain performance gains with buffer replication. The reason is that even though buffer replication breaks an anti dependence between two tasks caused by accesses to a buffer, there may exist other true dependences between these two tasks that prevent their parallel execution. Thus, in order not to introduce the run-time overhead of buffer replication caused by the helper tasks, we opt



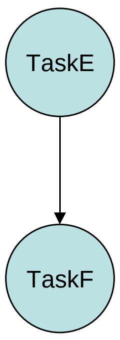
(a) FDG of `buf1`.



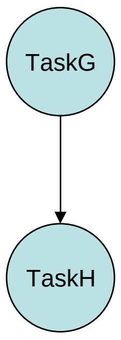
(b) ADG of `buf1`.



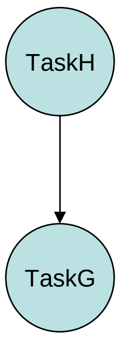
(c) FDG of `buf2`.



(d) ADG of `buf2`.



(e) FDG of `buf3`.



(f) ADG of `buf3`.

Figure 6.22: FDGs and ADGs for the running example buffer webs.

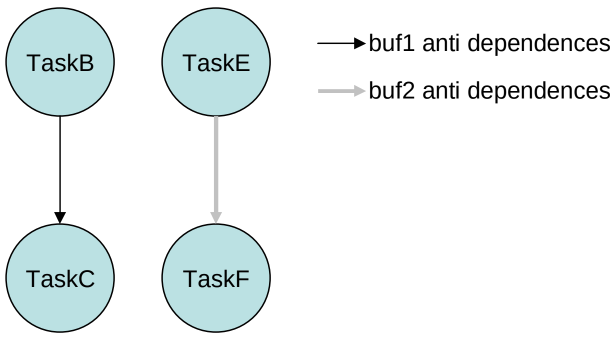


Figure 6.23: Eligible buffer replication candidates for the running example.

not to apply buffer replication to an anti dependence, if the sink and source tasks of the dependence are flow-dependent to each other.

For the running example with ADGs of Figure 6.22, the replication candidate of `buf3`, i.e. the anti-dependence from `TaskH` to `TaskG`, is eliminated, because `TaskG` is dependent to `TaskH` due to accesses to scalar URF variable `count`. On the other hand, the anti-dependences of `buf1` and `buf2` do not contain serialized task calls, therefore they are eligible for buffer replication. The eligible buffer replication candidates for the running example are shown in Figure 6.23.

6.7.2 Buffer Replication in the Control Program

For each eligible anti dependence, with source node $\mathtt{T}_1$ and sink node $\mathtt{T}_2$, of each buffer web `buf`, buffer replication is performed in four steps:

1. A private copy `pri` of `buf` is created.

2. `pri` is initialized with `buf`.

3. `buf` arguments of $\mathtt{T}_1$ are renamed to `pri`.

4. `pri` is destroyed after $\mathtt{T}_1$.

Buffer replication is performed with insertion of three helper tasks.

Init: Similar to buffer privatization, an `Init_R` task is inserted, just before the source node of the target anti dependence, to create the copy of the buffer to be replicated. This `Init_R` task has a single output argument, i.e. copy buffer, and no input argument. Its format is as follows:

`Init_R_n(out buffer_r_n); where n is the unique buffer replication id.`

Copy: A `Copy_R` task is inserted just after the `Init_R` task to initialize the copy buffer with data in the replicated buffer. This `Copy_R` task has two input and two output arguments: the copy buffer and the replicated buffer. Its format is as follows:

`Copy_R_n(in buffer_r_n, in buf, out buffer_r_n, out buf); where n is the unique buffer replication id and buf is the replicated buffer.`

Finish: A `Finish_R` task is inserted after the source node of the target anti-dependence to destroy the no longer needed copy. This `Finish_R` task has one input argument, i.e. copy buffer, and the following format:

`Finish_R_n(in buffer_r_n); where n is the unique buffer replication id.`

Figure 6.24 depicts the resulting output control program after buffer replication is applied to the input control program of the running example.

6.7.3 Creating Helper Task Functions

After the replication in the control program is completed, the function bodies of the helper tasks, i.e. `Init_R_n`, `Copy_R_n` and `Finish_R_n`, should be generated according to the type of the replicated buffer. For this purpose, the type of the replicated buffer is taken from the allocation node of the buffer. Then, a `malloc` operation in `Init_R_n`, a `memcpy` operation in `Copy_R_n` and a `free` operation in `Finish_R_n` are performed in order to create, initialize and destroy the replica of the buffer.

```
// Allocates buf1, buf2 and buf3 of type int(*)[21]
// Initializes count of type int
Init(out buf1, out buf2, out buf3, out count);

// Defines buf1[0:10]
TaskA(in buf1, out buf1);

while()
{
  Init_R_0(out buffer_r_0);

  Copy_R_0(in buffer_r_0, in buf1, out buffer_r_0, out buf1);

  // Uses buffer_r_0[0:20]
  TaskB(in buffer_r_0, out buffer_r_0);

  Finish_R_0(in buffer_r_0);

  // Defines buf1[11:20]
  TaskC(in buf1, out buf1);

  // Uses buf2[0:20]
  // Defines buf2[0:20]
  TaskD(in buf2, out buf2);

  Init_R_1(out buffer_r_1);

  Copy_R_1(in buffer_r_1, in buf2, out buffer_r_1, out buf2);

  // Uses buffer_r_1[0:20]
  TaskE(in buffer_r_1, out buffer_r_1);

  Finish_R_1(in buffer_r_1);

  // Uses buf2[0:20]
  // Defines buf2[0:20]
  TaskF(in buf2, out buf2);

  // Defines buf3[0:20]
  TaskG(in buf3, in buf3, in count);

  // Uses buf3[0:20]
  TaskH(in buf3, in buf3, out count);
}

// DeAllocates buf1, buf2 and buf3
Finish(in buf1, in buf2, in buf3);
```

Figure 6.24: The output control program of buffer replication.

For the example buffer replication in Figure 6.24, the `Init_R`, `Copy_R` and `Finish_R` task bodies shown in Figure 6.25 are created for `buf1` and, similarly, for `buf2`.

6.8 Buffer Renaming

As it is discussed in Section 5.3, buffer renaming aims to solve synchronization false dependences introduced by the MLCA programming model.

By definition, a synchronization false dependence exists when two tasks $\texttt{T}_1$ and $\texttt{T}_2$ do not have any memory dependence (i.e. flow, output and anti) with each other, for a buffer `buf` in the memory, but a URF dependence in the control program, caused by accesses to the pointer of `buf`, prevents the parallel execution of $\texttt{T}_1$ and $\texttt{T}_2$. With buffer renaming, for each buffer web, the graph of synchronization false dependences is obtained and each synchronization false dependence is solved separately by means of renaming the arguments of the dependent tasks.

A *synchronization false dependence graph (SFG)* is a directed multi-graph $G_s = (V, E_s)$ whose vertices V are the task calls of the control program and the edges E_s are the synchronization false dependences among task calls. Therefore, in the SFG that belongs to a buffer web `buf`, there exist an edge from $\texttt{T}_1$ to $\texttt{T}_2$, only if all the below conditions are satisfied.

1. If $\texttt{T}_2$ is not data-dependent on $\texttt{T}_1$, in terms of accesses to all the memory buffers and URF registers in the control program, i.e. there exists no path from $\texttt{T}_1$ to $\texttt{T}_2$ in the multi-variable FDG of the control program.

2. No memory false dependence exists from $\texttt{T}_1$ to $\texttt{T}_2$, in terms of accesses to the buffer referred by `buf`, i.e. there exist no path from $\texttt{T}_1$ to $\texttt{T}_2$ in the ADG and ODG of `buf`.

3. An element of `buf` is an output argument of $\texttt{T}_1$ and an input argument of $\texttt{T}_2$.

```
int Init_R_0()
{
  int (*copy_buffer)[21];

  copy_buffer = malloc(sizeof(int) * 21);

  writeArg(0, copy_buffer);

  return 1;
}
```

(a) Task function of the `Init_R_0`.

```
int Copy_R_0()
{
  int (*copy_buffer)[21];
  int (*original_buffer)[21];

  copy_buffer = readArg(0);
  original_buffer = readArg(1);

  memcpy(copy_buffer, original_buffer,
         sizeof(*copy_buffer));

  writeArg(0, copy_buffer);
  writeArg(1, original_buffer);

  return 1;
}
```

(b) Task function of the `Copy_R_0`.

```
int Finish_R_0()
{
  int (*copy_buffer)[21];

  copy_buffer = readArg(0);

  free(copy_buffer);

  return 1;
}
```

(c) Task function of the `Finish_R_0`.

Figure 6.25: Task functions of the buffer replication helper tasks for the running example.

Synchronization false dependences are solved using the SFG in two steps.

First, for each buffer web `buf_web` and for each task call $\texttt{T}_1$ in the SFG of `buf_web`, if there exists an edge originating from $\texttt{T}_1$, for every output argument `arg` of $\texttt{T}_1$, which is an element of `buf_web`, `arg` is renamed to an artificially created Sarek variable `arti_arg_n`, where `n` is a unique buffer renaming index. In other words, by renaming the synchronization argument causing the flow dependence between two task calls which, in fact, can execute in parallel, the false dependence between the two tasks is eliminated, enabling their parallel execution.

Second, new synchronization false dependences are created between $\texttt{T}_1$ and other tasks $\texttt{T}_2$, $\texttt{T}_3$, ... $\texttt{T}_n$ that are memory dependent (flow, output, anti) on $\texttt{T}_1$ in terms of accesses to `buf`; such that the new artificial argument `arti_arg` is declared as input arguments of $\texttt{T}_2$, $\texttt{T}_3$, ... $\texttt{T}_n$. This is necessary, because after the renaming, none of tasks that are accessing the buffer `buf`, referred by `buf_web`, will serialize with $\texttt{T}_1$, possibly violating true and false memory dependences. The reason is the lack of synchronization output arguments to schedule $\texttt{T}_1$ in serial with other tasks accessing `buf`. In addition, in order to serialize $\texttt{T}_1$ also with the task $\texttt{T}_d$ that is deallocating `buf`, `arti_arg` is also declared as an input argument of $\texttt{T}_d$.

Furthermore, since buffer renaming breaks all the synchronization false dependences originating from $\texttt{T}_1$, the other synchronization false dependences originating from $\texttt{T}_1$ do not need to be processed. In other words, after the renaming of `arg` with `arti_arg` in $\texttt{T}_1$, none of the tasks that are accessing `buf` will serialize with $\texttt{T}_1$, breaking other, if any, synchronization false dependences originating from $\texttt{T}_1$. Consequently, for each task call `T` in the SFG only one synchronization false dependence originating from `T` is solved, as the others will be automatically solved.

The algorithm of buffer renaming is shown in Figure 6.26.

It is crucial to note that, in the MLCA programming model, no exception is generated when a task's input argument that is not defined previously, i.e. not written to URF

```
for each buffer web buf_web
  for each vertex v1 in the SFG
    if there exists an edge that has v1 as its source
      increment buffer renaming index n
      for each output argument arg of buf_web in v1
        rename arg to arti_arg_n
      for each FDG, ODG, ADG of buf_web
        for each path from v1 to a vertex vp
          insert arti_arg_n to input arguments of vp
      for each deallocation task vd of buf_web
        insert arti_arg_n to input arguments of vd
```

Figure 6.26: The algorithm of buffer renaming.

```
TaskA(out value);

TaskB(in value, in count);

TaskC(out count);
```

Figure 6.27: No exception is generated in `TaskB`.

previously, is read from the URF. For example, the control program in Figure 6.27 will not produce any exception for `TaskB`, even though it reads `count` Sarek variable from the URF, before `count` is defined in `TaskC`.

Furthermore, no exception is generated when a task `T` has an nth input argument `arg` specified in the control program, but this nth input argument is not read from URF in the task function with a `readArg` routine. On the other hand, since URF dependences are processed by the CP according to only control program but not task functions, any dependence on `arg` will not be violated, i.e. false dependences will be resolved through renaming and true dependences will be satisfied by serializing dependent tasks. For example, in the control program and the corresponding task functions shown in Figure 6.28, `TaskB` will be serialized with `TaskA` although `TaskB` does not read the dependence causing argument `count` in its task function; further, no exception will be generated when `TaskB` executes.

In the light of above discussions, it can be concluded that buffer renaming is a valid transformation and requires no modification on the task functions of the modified task

```
TaskA(out count);

TaskB(in count);
```

(a) The example control program.

```
int TaskA()
{
  int var = 0;
  write_Arg(0, var);
  return 1;
}
```

(b) The body of `TaskA`.

```
int TaskB()
{
  return 1;
}
```

(c) The body of `TaskB`.

Figure 6.28: `TaskB` and `TaskC` are serialized and no exception is generated when `TaskB` executes.

calls.

Buffer renaming is illustrated with the example control program shown in Figure 6.29. In the example control program, for simplicity, the results of the allocation, deallocation, section definition and section use analyses of the task functions are shown as comments to task calls. In addition, `buf_1` variable is declared as both output and input arguments of the tasks it appears in, in order to synchronize tasks that are accessing the buffer according to the true and false dependences. This may be the case when the control program is directly generated by the programmer or by the previous code transformations such as parameter deaggregation, buffer privatization and buffer replication.

Moreover, although buffer privatization code transformation can be applied for `buf_1`, for simplicity, we will assume that it is not applicable.

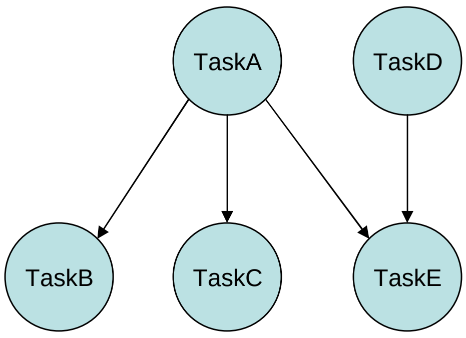
```
// Allocates buf_1
Init_1(out buf_1);

while(cont)
{
  // Defines buf_1[0:10]
  TaskA(in buf_1, out buf_1);

  // Uses buf_1[0:10]
  TaskB(in buf_1, out buf_1);

  // Uses buf_1[0:10]
  TaskC(in buf_1, out buf_1, out count);

  // Defines buf_1[11:20]
  TaskD(in buf_1, out buf_1);

  // Uses buf_1[0:20]
  TaskE(in buf_1, in count, out buf_1);
}

// Deallocates buf_1
Finish_1(in buf_1);
```

Figure 6.29: The example control program for buffer renaming.

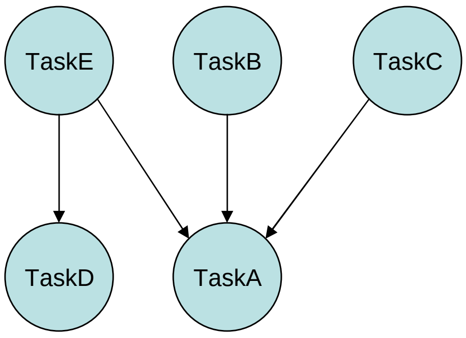
Furthermore, `buf_1` Sarek variable represents the only buffer web in the control program, thus, it will be used to represent the referred memory buffer. Figure 6.30 depicts the FDG, ADG and ODG of `buf_1`. Since `TaskA` and `TaskD` define distinct regions of `buf_1`, the ODG of `buf_1` contains only self-edges on `TaskA` and `TaskD`.

Apart from `buf_1`, there also exists a `count` Sarek variable which is of scalar type, in the example control program. Consequently, the union of FDGs of `count` and `buf_1` compose the multi-variable FDG of the control program, which is shown in Figure 6.31.

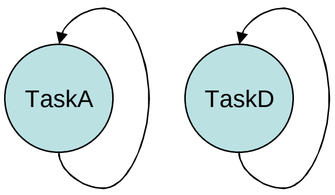
Using the multi-variable FDG of the control program, FDG, ADG and ODG of `buf_1`, the SFG of `buf_1` is generated as shown in the Figure 6.32. In the control program of the running example, `TaskC` can execute in parallel with `TaskB`, as there is no path from `TaskB` to `TaskC` in the multi-variable FDG and in any ADG and ODG of all the buffers. However, they are serialized because `buf_1` is declared as an output argument of `TaskB` and an input argument of `TaskC`. Consequently, there is an edge from `TaskB` to `TaskC` in the SFG of the `buf_1`. In addition, `TaskC`, in one iteration of the loop, can execute in parallel with the `TaskB` of the next iteration, because there is no path from `TaskC` to



(a) FDG of `buf_1`.



(b) ADG of `buf_1`.



(c) ODG of `buf_1`.

Figure 6.30: FDG, ADG and ODG of `buf_1` for the running example.

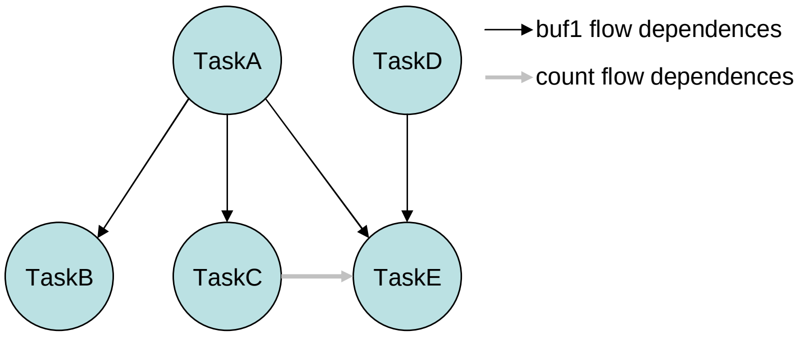


Figure 6.31: The multi-variable FDG for the running example.

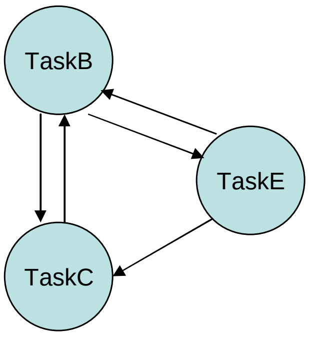


Figure 6.32: SFG of the control program for the running example.

TaskB in the above mentioned dependence graphs. Nevertheless, the fact that buf_1 is declared as an output argument in TaskC and as an input argument in TaskB prevents this parallel execution. For this reason, there is also an edge from TaskC to TaskB in the SFG. Similarly, there exist edges between TaskB and TaskE. However, there exist only a single edge between TaskC and TaskE, because TaskE can not execute in parallel with TaskC in the same iteration of the loop because of the scalar flow dependence caused by the count Sarek variable seen in the multi-variable FDG. On the other hand, TaskC in one iteration can execute in parallel with TaskE of the previous iteration, as there is no path from TaskE to TaskC in the multi-variable FDG.

When the algorithm of buffer renaming is applied to the SFG of buf_1, buf_1 is renamed in the output arguments of all vertices of SFG, i.e. TaskB, TaskC and TaskE. In TaskB, the output argument buf_1 is renamed to buf_arti_1 in order to solve the synchronization false dependence from TaskB to TaskC and TaskE. However, since the synchronization dependence from TaskB to TaskA, which is satisfying the anti-dependence between them, is also broken by this renaming, buf_arti_1 is also made an input argument of TaskA. Similarly, buf_1 output arguments of TaskC and TaskE are renamed to buf_arti_2 and buf_arti_3 respectively, breaking the synchronization false dependences, and buf_arti_2 and buf_arti_3 are made input arguments of TaskA in order to satisfy the anti-dependences from TaskC and from TaskE to TaskA. Furthermore, the artificial arguments buf_arti_1, buf_arti_2 and buf_arti_3 are made input arguments of Finish, which deallocates the buffer buf_1 and, thus, should execute after all the tasks accessing buf_1 are complete.

Figure 6.33 depicts the control program obtained as the result of buffer renaming.

```
// Allocates buf_1
Init_1(out buf_1);

while(cont)
{
  // Defines buf_1[0:10]
  TaskA(in buf_1, out buf_1,
        in buf_arti_1, in buf_arti_2, in buf_arti_3);

  // Uses buf_1[0:10]
  TaskB(in buf_1, out buf_arti_1);

  // Uses buf_1[0:10]
  TaskC(in buf_1, out buf_arti_2, out count);

  // Defines buf_1[11:20]
  TaskD(in buf_1, out buf_1, in buf_arti_3);

  // Uses buf_1[0:20]
  TaskE(in buf_1, in count, out buf_arti_3);
}

Finish_1(in buf_1,
         in buf_arti_1, in buf_arti_2, in buf_arti_1);
```

Figure 6.33: The control program after buffer renaming.

Chapter 7

Compiler Design

In this chapter, we present the MLCA Optimizing Compiler (MOC). The design criteria for the MOC are discussed and an overview of its architecture is given. Section 7.1 presents the overall design of the MOC. Section 7.2 presents the Sarek pragmas, which are the medium of communication between different compilation phases, and between the programmer and the MOC.

7.1 The MLCA Optimizing Compiler

In this section, we discuss the architecture of the MOC. First, we present the architecture, then, we justify this architecture based on the features of our code transformations and the Sarek language. Finally, we present the benefits of this architecture.

The MOC is responsible of optimizing the performance of its input control program together with the corresponding task functions, in terms of total execution time. This is achieved by applying the Sarek code transformations described in Chapter 5, i.e. parameter deaggregation, buffer privatization, buffer replication, buffer renaming and code hoisting.

MOC is designed to be a system of two sub-compilers, a *C-Compiler* and a *Sarek-Compiler*, processing two different languages in a single run, as depicted in Figure 7.1.

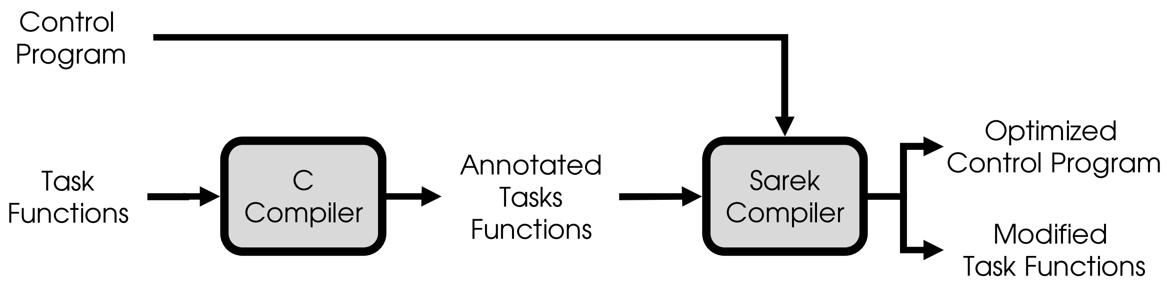


Figure 7.1: The architecture of the MLCA Optimizing Compiler.

The **C-Compiler** takes the task functions and other helper functions of the application as input and applies compiler analyses such as inter-procedural array-section analysis and inter-procedural data-flow analysis of the structure fields. The results of these analyses, together with the types of the task arguments are sent to the Sarek-Compiler.

The **Sarek-Compiler** takes the input control program and the results of the task function analyses produced by the C-Compiler as input. It applies the Sarek code transformations to the input control program, optimizes the control program and modifies the task functions accordingly.

The communication between the C-Compiler and the Sarek-Compiler is achieved with annotations inserted in the task functions. The results of the analyses performed by the C-Compiler are inserted in the code of the task functions in the form of pragma statements. The Sarek-Compiler takes the annotated task functions as input and retrieves the pragma annotations. It applies the Sarek code transformations using these results and modifies the control program and the task functions accordingly.

An API is also provided for the pragma annotations, which allows the programmers to directly insert data usage information in the code of the task functions. Pragmas inserted by the programmer override the pragma annotations generated by the C-Compiler; hence, the programmer can modify the information supplied to the Sarek-Compiler.

The design of the MOC is based on the fact that it is not possible to consider a control program apart from the task functions, consequently a Sarek-Compiler from a

C-Compiler. In the remainder of this section, we justify this with three facts.

First, Sarek, by being a high level language, is designed to represent the inter-procedural data and control flow of an application. It is not involved in any computation, i.e. it does not perform or represent any work; rather, it schedules tasks which are the work functions. Therefore, in order to process the work of the tasks, such as memory accesses, variable definition/use, etc., control programs are not sufficient sources of information and, in fact, task functions are needed to be analyzed.

Second, Sarek does not include strong typing, as each of its register variables (`reg_t`) represents data of fixed size, which can either be a scalar value or a pointer. Consequently, it is not possible to distinguish the types of the task arguments from the control program point-of-view. Therefore, the inspection of the task functions for the types of their input/output variables is necessary.

Third and more significantly, the Sarek code transformations can not be applied without analyzing or modifying the task functions, because of the following reasons:

1. Buffer privatization, buffer replication, buffer renaming and parameter deaggregation require the types of the task arguments to distinguish buffers and structures.
2. Buffer privatization, buffer replication and buffer renaming require the inter-procedural array-section analysis results for the task functions.
3. Parameter deaggregation requires inter-procedural data-flow analysis results for the structure fields.
4. Code-hoisting requires the intra-procedural data-flow analysis results for the task arguments inside the task functions.
5. Code-hoisting relocates the `writeArg` routines in the task functions.
6. Parameter deaggregation modifies the task arguments; thus, `writeArg` routines have to be altered accordingly inside the task functions.

7. Buffer privatization and buffer replication create new tasks, for which new task functions has to be generated.

The design of the MOC provides some important benefits.

- *Different Compiler Infrastructures:* Different compiler infrastructures for the C-Compiler and the Sarek-Compiler can be used. This provides the freedom of selecting the most suitable infrastructure for each sub-compiler and also replacing one sub-compiler without modifying the other one.

- *Ease of Development:* The sub-compilers can be developed separately. Thus, after one sub-compiler is developed and tested, the other one can be started. This will ease debugging during the development process, because in case of an unexpected transformation outcome, it is possible to isolate the failing sub-compiler.

- *Simple Sub-Compilers:* The Sarek-Compiler is only responsible of applying the Sarek code transformations and is not involved in any complex analyses of the task functions. Similarly, the C-Compiler only performs compiler analyses on the input functions and is not involved in any modification of the control program. In fact, the control program is not an input to the C-Compiler.

- *Easy Observation of the Information Flow:* The information flow from the C-Compiler to the Sarek-Compiler can be observed by the programmer by only inspecting the output task functions of the C-Compiler. Hence, adjustments about the conservativeness of the C-Compiler can be made easily.

- *Simple Control Program:* The fact that the information flow from the C-Compiler to the Sarek-Compiler is through the task functions keeps the control program simple and allows independent development of the task functions. In other words, after the control program is generated for an application, task functions can be developed independently, as long as the input and the output arguments are consistent with the control program. Thus, any change in the implementation of a task function will be reflected on the pragmas inside the task function, but not on the control program.

- *Programmer Control Over Compilation:* The API for the pragma annotations provides to the programmer the ability to control the compilation of the control program. Any missing and/or incorrect analysis results can be replaced by the programmer, who has a reasonable understanding of the application's functionality or by a profiler that has run-time profiling information of the application. Considering that some of the analyses required by the Sarek code transformations, such as array-section analysis, are complex compiler analyses that do not have successful implementations in the literature and are dependent on the run-time behavior of the applications, the feedback of a programmer or a profiler is crucial for the MOC. In fact, in most cases, a section definition/use in a task function, which can easily be observed by the programmer with the inspection of the code, may not be impossible for the compiler to produce due to I/O operations and control-flow decisions that can not be predicted at compile-time.

7.2 Sarek Pragmas

The Sarek pragmas are special comments that are identified with a unique sentinel. The syntax of the user API for the Sarek pragmas is as follows:

```
sentinel directive_name item1 [item2] [identifier1] [identifier2]
```

The scope of the pragmas is the task function they appear in and their functionality is independent of their location inside the function. The following sections describe the types and functionalities of the Sarek pragmas.

7.2.1 Items

Items are the affected targets of the pragmas. For the task and clone pragmas, the item is a task name, while for others, it is either a structure field accessed via a local structure pointer, or a local variable which points to a buffer or a structure. For variables, variable name is used to describe the item, whereas for the structure fields, the offset of the field with respect to the local structure pointer is given, using "`->''`or ".", in the same way the field is accessed in a C expression.

In case the item is a buffer pointer, it may be followed by a region identifier to specify a region of the buffer.

7.2.2 Region Identifier

The region identifier describes a region for an item. Two real numbers define the start and the end offset values for the region. The syntax of the identifier is as follows:

```
[start_offset:end_offset]
```

For instance, `buf[20:40]` represents the buffer region between 20^{th} and 40^{th} elements (inclusive) of the buffer (or the buffer pointed by) `buf`.

Optionally, "*" wildcard can be used to describe a full region for an item, spanning the whole buffer. For example `buf[*]` represents all the elements of the buffer `buf`.

The regions for arrays with multiple dimensions are represented as offsets from the start address of the array.

7.2.3 Task Pragma

The Task pragma defines a task function for a specified task of the control program. The task function is the function that the pragma is in and the task name is given as the item of the pragma.

The syntax of the Task pragma is as follows:

```
#pragma mlca Task task_name

function_header

start_of_function

[ statements ]

end_of_function
```

The `task_name` is the name of the task that the task function belongs to.

7.2.4 Clone Pragma

When two tasks have the same function as their task functions, it is said that the two tasks are *clones* of each other. In case that multiple tasks are clones of each other, one task is defined as the primary task using a Task pragma and the other tasks are declared as the clones of the primary task using Clone pragmas.

Since a single function may be the task function for multiple tasks, there may exist multiple clones of a task and therefore multiple Clone pragmas for a single function.

The Clone pragma has the following syntax:

```
#pragma mlca Task task_name

#pragma mlca Clone clone_task_name clone_id

function_header

start_of_function

[ statements ]

end_of_function
```

The `task_name` is the name of the primary task for the task function, whereas the `clone_task_name` is the name of the clone task for the task function.

Each clone task is given a unique `clone_id` to be used in Definition, Non-Definition, Use and Non-Use pragmas. In fact, when a `clone_id` is not given, these pragmas are effective for the primary task and also for all the clone tasks. When a `clone_id` is given, they are only effective for the clone task with the id `clone_id`.

7.2.5 Allocation Pragma

The syntax of the Allocation pragma is as follows:

```
#pragma mlca Alloc item [ allocation_size ]
```

The Allocation pragma declares that a memory location pointed by the `item` is allocated inside the function that the pragma is attached to.

The following conditions should be specified for an item to be declared as allocated:

1. The item should be of type pointer.
2. The item should be an output variable.
3. The item should carry the value of the pointer for the allocated memory in every `writeArg` routines it appears.

In cases where the item is of type pointer to scalar, the allocation size in bytes should be identified with `allocation_size`.

7.2.6 Deallocation Pragma

The syntax of the Deallocation pragma is as follows:

```
#pragma mlca DeAlloc item
```

The Deallocation pragma declares that a memory location pointed by the specified `item` is destroyed in the function that the pragma is in.

In order for an item to be declared as deallocated, the following conditions should be met:

1. The item should be of type pointer.
2. The item should be an input argument.
3. The item should carry the value of the pointer to the shared memory location destroyed, in the `readArg` routine it is assigned in.

7.2.7 Definition Pragmas

The general syntax of the Definition pragma is as follows:

```
#pragma mlca Defs item [ identifier ] [ Clone clone_id ]
```

The Definition pragma declares that the specified `item` is defined in the task function. The `item` should be an input argument of the task function that the pragma is in.

The types of the items and the identifiers differ according to the meaning of the Definition pragma. In fact, the Definition pragmas are divided into two in terms of the types of their items and identifiers.

Structure Definition Pragma

The format of the Structure Definition pragma is as follows:

```
#pragma mlca Defs item [ * ] [ Clone clone_id ]
```

The Structure Definition pragma is used to declare that the specified `item`, which is either a local structure pointer (input argument of task function) or a structure field accessed with a local structure pointer (input argument of task function), is defined in the task function the pragma is in. If the definition of the item does not occur in every invocation of the task function, the item should also be declared as used with a Use pragma in addition to being declared as defined. This situation is discussed in Section 7.2.12.

In cases where the item is of type pointer-to-structure, the "`*`" identifier marks every field in the pointed structure as defined.

Section Definition Pragma

The format of the Section Definition pragma is as follows:

```
#pragma mlca Defs item region_identifier [ Clone clone_id ]
```

The Section Definition pragma is used to declare that the region specified by the `region_identifier` of the buffer specified by the `item`, which is either of type pointer-to-scalar or pointer-to-buffer, is defined in the task function that the pragma is in.

In order for a region to be declared as defined, all the elements of the region should be defined in at least one invocation of the task. In that sense, the Section Definition pragma considers the union of all the regions (of a specific buffer) defined by a task. In case there exists a region of the buffer which is defined in some invocations of the task function, but not in all of its invocations, then this region should also be declared as used with a Use pragma, in addition to being declared as defined. This situation is discussed in Section 7.2.12.

In case, several Section Definition pragmas are provided for the same item, the specified regions are merged.

7.2.8 Non-Definition Pragmas

The general syntax of the Non-Definition pragma is as follows:

```
#pragma mlca NoDefs item [ identifier ] [ Clone clone_id ]
```

The Non-Definition pragma is used to declare that the `item` specified is not defined in the task function. The item should be an input argument of the task function that the pragma is in.

This pragma is not generated by the C-compiler. It is designed to allow the programmer to eliminate the definition pragmas generated by the C-compiler.

The types of the items and the identifiers differ according to the meaning of the Non-Definition pragma. In fact, the Non-Definition pragmas are divided into two in terms of the type of their items and identifiers.

Structure Non-Definition Pragma

The format of the Structure Non-Definition pragma API is as follows:

```
#pragma mlca NoDefs item [ * ] [ Clone clone_id ]
```

The Structure Non-Definition pragma is used to declare that the specified item, which is either a local structure pointer (input argument of task) or a structure field accessed with a local structure pointer (input argument of task), is not defined in any execution path of the task function that the pragma is in. In other words, Non-Definition pragma states that, not a single definition to the item exists, in any reachable location of the procedure. It is used to eliminate the effect of any Structure Definition pragma for the specified item.

In cases where the item is of type pointer-to-structure, the "*" identifier marks every field in the memory storage pointed by the item as not-defined.

Section Non-Definition Pragma

The format of the Section Non-Definition pragma is as follows:

```
#pragma mlca NoDefs item [ region_identifier ] [ Clone clone_id ]
```

The Section Non-Definition pragma is used to declare that the region specified by the `region_identifier` of the buffer specified by the `item`, which is either of type pointer-to-scalar or pointer-to-buffer, is not defined in any execution path of the procedure that the pragma is in. In fact, Section Non-Definition pragma eliminates any definition declaration for the specified region of the specified item throughout the task function.

7.2.9 Use Pragmas

The syntax of the Use pragma is as follows:

```
#pragma mlca Uses item [ identifier ] [ Clone clone_id ]
```

The Use pragma marks an `item` as used for the task function that the pragma is in. The `item` should be an input argument of the task function that the pragma is in.

The types of the items and the identifiers differ according to the meaning of the Use pragma. In fact, the Use pragmas are divided into two in terms of the types of their items and identifiers.

Structure Use Pragma

The format of the Structure Use pragma is as follows:

```
#pragma mlca Uses item [ * ] [ Clone clone_id ]
```

The Structure Use pragma is used to declare that the specified `item`, which is either a local structure pointer (input argument of the task) or a structure field accessed with a local structure pointer (input argument of the task), is used in the task function the pragma is in. The Structure Use pragma requires that at least a single use of the specified item dominates every definition to the item or there exist definitions to the `item` which do not occur in every invocation of the task function.

In cases where the item is of type pointer-to-structure, the "*" identifier marks every field in the memory storage pointed by the item as defined.

Section Use Pragma

The format of the Section Use pragma is as follows:

```
#pragma mlca Uses item region_identifier [ Clone clone_id ]
```

The Section Use pragma is used to declare that the region specified by the `region_identifier` of the buffer specified by the `item`, which is either of type pointer-to-scalar or pointer-to-buffer, is used in the task function that the pragma is in.

The Section Use pragma further states that section uses of the region for the item dominates every definition of the same section or definitions to the section of the item does not occur in every invocation of the task function. If different sections of an item are used in distinct invocations of the task function then, the union of these sections is declared as used.

In case, several Section Use pragmas are provided for the same item, the specified regions are merged.

7.2.10 Non-Use Pragmas

The syntax of the Non-Use pragma is as follows:

```
#pragma mlca NoUses item [ identifier [Clone clone_id] ]
```

The Non-Use pragma marks an `item` as not used for the task function that the pragma is in. It eliminates the effect of any Uses pragma for the same item in the task function. The item should be an input argument of the task function that the pragma is in.

This pragma is not generated by the C-compiler. It is designed to allow the programmer to eliminate the Use pragmas generated by the C-compiler.

The types of the items and the identifiers differ according to the meaning of the Non-Use pragma. In fact, the Non-Use pragmas are divided into two in terms of the type of their items and identifiers.

Structure Non-Use Pragma

The format of the Structure Non-Use pragma is as follows:

```
#pragma mlca NoUses item [ * ] [ Clone clone_id ]
```

The Structure Non-Use pragma is used to declare that the specified `item`, which is either a local structure pointer or a structure field accessed with a local structure pointer, is not used in any execution path of the task function that the pragma is in. In other words, the Structure Non-Use pragma states that, either not even a single use of the item exists or every use of the item is dominated by a definition to it, and every definition to the item occurs in every invocation of the task function. In fact, Non-Use Pragma is used to eliminate the effect of any Structure Use pragma for the specified item.

In cases where item is of type pointer-to-struct, the "*" identifier marks every component in the memory storage pointed by the item as un-defined.

Section Non-Use Pragma

The format of the Section Non-Use Pragma API is as follows:

```
#pragma mlca NoUses item [ region_identifier ] [ Clone clone_id ]
```

The Section Non-Use pragma is used to declare that the region specified by the `region_identifier` of the buffer specified by the `item`, which is either of type pointer-to-scalar or pointer-to-buffer, is not used in any execution path of the function that the pragma is in. In other words, Section Non-Use pragma states that either the specified region is not used in any part of the function or every use of the section is dominated by a definition to the same section and every definition to the region of the item is defined in every invocation of the task function. In fact, Section Non-Use pragma effectively eliminates any use declaration for the specified region of the specified item throughout the task function that the pragma is in.

7.2.11 Alias Pragma

The general syntax of the Alias pragma is as follows:

```
#pragma mlca Alias item1 item2 identifier
```

The Alias pragma declares an alias relationship between `item1` and `item2`, which are of type pointer. `item2` should be an output argument, whereas `item1` can either be an input or output argument of the task.

The `identifier` may be one the following three:

simple states that `item2` points to the same exact location as `item1` at the exit of the task function.

complex states that `item1` and `item2` may be aliases at the exit of the task function, but whether they point to the same location can not be determined.

none states that `item2` is not alias with `item1` at the exit of the task function.

7.2.12 Declaring Data as Input

In the MLCA programming model, input and output arguments of tasks are unconditional. In other words, control processor expects a task to read and write all of its input and output arguments in every invocation of the task. A variable should be made an output argument of the task, if it is defined in the task. However, in some cases, a variable which is an output argument of a task may not be defined in every invocation of the task because of control flow paths taken at run-time. This variable should also be declared as an input argument of the task, even if it is not explicitly used in the task. Thus, the variable will carry the value stored in the URF during the execution of the task and, in case the variable is not defined, the correct value will be written to URF.

Figure 7.2 depicts such a case, in which the output argument `arg` is also made an input argument of the task, even though it is not used in the task. Hence, when `arg` is not defined in the task (i.e. if condition is false), correct value of `arg` will be written to the URF.

```
int Task()
{
  int arg, b;

  arg = readArg(0);

  b=...

  if(b < 0)
   arg = ...

  writeArg(arg);

  return 1;
}
```

Figure 7.2: `arg` is also an input argument because it is not always defined in the task.

The described situation is also valid for the structure fields and buffer sections. Consequently, in case a task does not always define a field of a structure, this structure field should be marked as used (with a Use pragma) in addition to being declared as defined (with Definition pragma). Similarly, when a section of a buffer is not always defined inside a task, this section should also be marked as used in addition to being marked as defined.

Chapter 8

Experimental Evaluation

In this chapter, we evaluate the performance of the MLCA compiler using real multimedia applications. Section 8.1 introduces our experiment platform. Section 8.2 describes the multimedia applications ported to MLCA and used as benchmarks in the experiments. Section 8.3 presents our methodology. Section 8.4 evaluates the performance of the MLCA Optimizing Compiler using manually inserted pragmas. Section 8.5 discusses the success of the C-Compiler in generating the required pragmas to the Sarek-Compiler.

8.1 Experiment Platform

We use a timed functional model to simulate the performance of the MLCA. The model consists of 6,000 lines of C++/SystemC and reflects the overall structure of the MLCA, with a Control Processor, Task Dispatcher, Universal Register File, some PUs, and shared memory.

The model instantiates the desired configuration at runtime. Parameters include: number and type of PUs, URF size, number of renaming registers, memory configuration and associated latencies, relative speed of CP, TD and PUs.

The model uses ARM processors for PUs. Each PU can be configured with a combination of local and global memory. The interconnect adds a constant delay, and the memory model implements a simple contention mechanism, where the requests are enqueued in order and dequeued at a given rate. The simulator models the URF contention in similar way.

The model produces a number of statistics, including: the length of the simulation, the number of instructions for each PU, number of read/write for the URF, the memories, number of cycles spent waiting for I/O, average latency for the memory operations, etc.

There also exists a simple tool to compile Sarek to HASM. The tool does not perform any optimizations, but its functionality is sufficient to avoid writing assembly-level code when applications are ported manually. The tasks themselves are compiled for ARM using the linux-to-ARM cross-compiler 3.2.2 version of GNU's GCC, *arm-elf-gcc*. The model loads into memory the ELF object file.

We run the model on a workstation which has two 2.0 GHz AMD Athlon MP 2400+ processors with 256 KB cache and 512 MB of memory.

8.2 The Applications

In this section, we describe the applications used as benchmarks in evaluating the performance of the MLCA compiler.

8.2.1 MAD

MAD [4] is an open source MPEG audio decoder that translates MPEG layer-3 (mp3) files into 16-bit PCM output. We use a stripped-down version of the code, which does not include multithreading, but retains the functionalities and code structure of the original application.

The input to MAD is a byte stream that represents a sequence of audio frames. Each frame consists of a frame header and frame data. The frame header contains configuration information such as audio layer type, channel mode, sampling frequency, stream bit rate, and the location of the frame's main data in the input stream. Since frames may be of different sizes, a frame header also contains the size of its corresponding frame.

The main data structure in MAD is a C structure called `mad_decoder`. It contains global variables and three other C structures: `mad_stream`, `mad_frame`, and `mad_synth`. The `mad_stream` structure stores the start and end addresses of the input stream in memory, a pointer to the start of the current frame being decoded, a pointer to the next frame to be decoded, and buffers used for decoding a frame. The `mad_frame` and `mad_synth` structures hold buffers for the decoded and the synthesized PCM output of a frame, respectively. Thus, most of the pointers and buffers within `mad_stream`, as well as within the `mad_frame` and `mad_synth` structures are re-used for the decoding of each frame.

MAD application first starts by allocating and initializing various data structures. Then, the file containing the input stream is mapped to memory, and the frames are decoded one at a time until end-of-file is reached. For each frame, the decoded output is copied to `mad_frame`. Next, the PCM output is synthesized and placed in the `mad_synth` structure. The structure is sent to either a file or the standard output. Finally, the input file is unmapped from memory, and the various structures are deallocated.

In our experiments, we run the MAD application to decode 64 frames, which takes 72.5 million cycles without any optimization.

8.2.2 FMR

FMR [3] is an open-source audio application that performs FM demodulation on a 16-bit input data stream, producing a 32-bit output data stream. The input stream consists of data packets of 1536 bytes each.

The main data structures used in the program are a set of buffers that are used to store and process each input packet. Pointers to these buffers are passed as arguments to the various functions.

The FMR application primarily performs a sequence of operations, such as CIC low pass filtering, FM demodulation, and IIR/FIR de-emphasis on each input packet to produce the output. These steps are performed in the main function by 70 calls to 16 different functions.

In our experiments, we run the FMR application to decode 22 input data packets, which takes 146.2 million cycles without any optimization.

8.2.3 GSM Encoder

The GSM encoder [5] is the open source implementation of the European GSM 06.10 provisional standard for full-rate speech transcoding, developed by the Technical University of Berlin. It uncompresses frames of 160 16-bit linear samples into 33-byte frames. The quality of the algorithm is good enough for reliable speaker recognition.

The main data structures of the GSM encoder consist of a structure named `gsm`, an input buffer, and an output buffer. The `gsm` structure contains several scalars that store various information about the encoding process, and buffers that store the intermediate results of the encoding. Some of these scalars and buffers are reused for the encoding of each frame, whereas some are shared between the encoding of subsequent frames.

The process of encoding a single GSM frame consist of six phases: preprocessing, linear predictive coding (LPC) analysis, short-term residual signal analysis, long-term predictor, regular pulse excitation (RPE) coding and frame formation. Each frame is taken from the input file and processed in these six phases, which are implemented in distinct functions. The output of encoding is stored to the output buffer and the output buffer is either written to a file or sent to the standard output.

In our experiments, we run the GSM application to decode 64 frames, which takes 33 million cycles without any optimization.

8.3 Methodology

We take three different approaches to port each benchmark application for the MLCA. First, for each application, we prepare a **baseline (B)** version of the application, in which only one task, that calls the main function of the application, is defined and, thus, the control program consists of a single task call to this task. A baseline version is the simplest version of an application that can run in MLCA; therefore, it does not include any overheads and is useful for comparing the impact of task selection and code transformations. Second, for each application, we prepare a **manually-optimized (MO)** version of the application, in which both the task selection as well as the code transformations are performed manually, aiming for the highest performance possible. Third, for each application, we prepare **compiler-optimized (CO)** versions of the application, in which the task selection is performed manually; but, the code transformations are applied by the MLCA compiler. Further, for each application several compiler-optimized versions are generated (with user-defined pragmas, without user-defined pragmas, with different code transformations enabled, etc.) leading to the experiments described in the remainder of this chapter.

We define the *base-speedup* as the ratio of the total execution cycles of an application (either manual-optimized or compiler-optimized) to the total execution cycles of the baseline version of the same application. Similarly, we define the *relative-speedup* as the ratio of the total execution cycles of an application (either manual-optimized or compiler-optimized) to the total execution cycles of the same version run on a single processor. Thus, base-speedup takes into account all the factors affecting the performance of the ported application, such as the overhead of the control processor and URF accesses, which are dependent on the architectural parameters. In contrast, the relative-speedup reflects only the impact of the code transformations on the total execution cycles of a control program and, hence, is used to evaluate the effectiveness of these transformations.

For all the benchmark applications, we select the control program and the corresponding tasks with a top-bottom approach. With this approach, the functions on the top of the application's call graph are first selected as tasks and the functions that are called by the current tasks are promoted to be the new tasks in later iterations of the task selection process. By following the same task selection approach in every experiment, we show that the compilation environment presented in Section 4.1 is effective in porting applications and it is possible to design a task selector that will work with the MLCA Optimizing Compiler.

We compiled the task functions and the helper functions of each benchmark application with the `O2` optimization level of the `arm-elf-gcc` and ran our experiments with an instance of MLCA which does not introduce any limitation on the maximum parallelism. In that sense, the number of renaming registers, the number of URF/memory ports, the depth of out-of-order execution and the shared/local memory sizes are chosen sufficient enough to obtain ideal speedups[1]. The architectural parameters of the MLCA instance used in our experiments are shown in Table 8.1.

Since the model is not fully capable of simulating cache behavior, in our experiments no caches are simulated.

In addition, the MLCA model introduces a limitation on control programs. For the model to correctly run the applications with multiple ARM processors, all the memory allocations and deallocations, should be run on a special processor, called *MemoryProc*, which has the same functionality as an ARM processor[2]. This is achieved, during the task selection, by grouping such memory operations in special tasks that the MemoryProc is assigned to. On the other hand, all the remaining tasks are run on a number of regular ARM processors, depending on the experiment. In fact, we present our results as a function of the number of regular ARM PUs. However, the mandatory processor assignment of memory allocation and deallocations hides their impact on the total execution cycles,

[1]The effect of these architectural parameters on the execution cycles of our benchmarks is outside the scope of this thesis and examined in our previous work [21].

[2]This limitation is not necessary for single ARM processor runs.

because with MemoryProc such memory operations may run in parallel with the rest of the application's instructions running on the ARM PUs. In fact, memory allocations and deallocations are the overhead of the buffer privatization and buffer replication code transformations presented in Chapter 5. Nonetheless, we also examine the impact of such memory operations (with one processor simulations), together with various other overheads, in the remainder of this chapter.

8.4 Sarek Compiler Performance

In this section, we present and report on the experiments on the speedups of the benchmark applications with the MOC, overheads affecting these speedups and the impact of the code transformations on the execution cycles.

8.4.1 Speedup Experiments

In order to evaluate the performance of the MOC and the effectiveness of the code transformations, we prepared compiler-optimized versions (in addition to the baseline and the manual-optimized versions) of each benchmark application with manually provided pragmas. These pragmas give the MOC complete buffer section definition/use and structure fields definition/use information and are obtained by manually inspecting the code of the corresponding application. Similarly, in order to enable the code transformations, the allocation and deallocation of each buffer and structure is also marked with Allocation and Deallocation pragmas inside the task functions.

Figure 8.1 shows the relative and base speedups for the manual-optimized and compiler-optimized versions of the MAD, FMR and GSM applications, with respect to a number of ARM PUs and a MemoryProc. In the figure, the trends of the relative and base speedups indicate the extracted parallelism. In addition, the starting points of the base-speedup curves indicate the overheads. However, the overheads of buffer privatization and buffer

Parameter	Value	Description
ARM_GLOBAL_START	0x20000	The start address of the shared memory. Everything below this address is considered as local memory, everything above as shared.
DELAY_CP_FETCH	1	CP fetches an instruction every DELAY_CP_FETCH cycles.
DELAY_INTERCONNECT	1	The delay for a read/write request to go through the interconnect.
NB_REG	5000	The number of logical registers in the URF.
OOO_DEPTH	1000	The depth of the out-of-order execution.
SIZE_CRF	16	The number of logical control registers in the Control Register File.
SIZE_CRF_KTB	100	The size of the renaming table for the Control Register File.
SIZE_KTB	5000	The size of the renaming table for the URF.
TD_QUEUE	1000	The size of the task dispatcher queue.
URF_LATENCY	1	The intrinsic latency of the URF for reading/writing one register.
URF_NB_PORTS	100	The number of read/write requests that can be processed concurrently in the URF.
MEMORY_LATENCY	1	The intrinsic latency of the memory for reading/writing one 32-bit word.
MEMORY_THROUGHPUT	1000	The number of 32-bit memory requests handled per cycle.

Table 8.1: The architectural parameters of the MLCA instance used in the experiments.

replication are hidden by the MemoryProc and, hence, are not reflected on the speedup curves, as is explained in the previous section. It is crucial to note that, CP and the task dispatcher plays a very important role during the execution and their impact on the execution cycles is dependent on the parameters of the MLCA instance experimented with. Therefore, with different MLCA instances or with a real MLCA architecture, the extracted parallelism, i.e. speedup trends, are expected to remain the same; on the other hand, the speedup values may change.

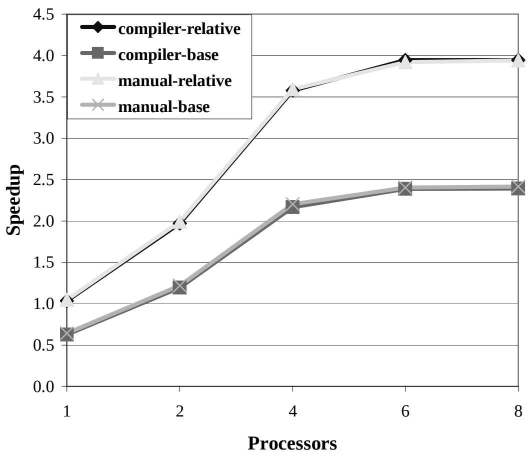
Performance Scalability

The manually and compiler optimized versions of all three applications exhibit scaling performance.

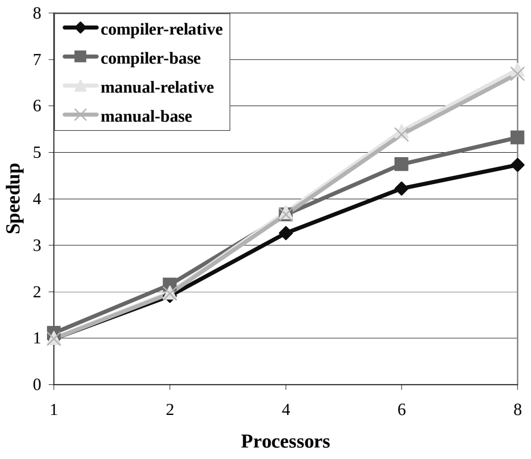
The speedup of the MAD application scales well up to 6 processors. With 8 processors, the available parallelism is fully exploited. Consequently, the relative speedup flattens at 3.9 and the base speedup flattens at 2.4. This upper limit for the performance is because of the loop-carried true dependences between the subsequent executions of a large task, which executes once for every input frame. Instances of this task (from different iterations of the loop) continues executing, even after all instances of all the remaining tasks are complete, prolonging the execution of the MAD application.

For the FMR application, the speedup scales well with 8 processors. Due to the increasing trend of the speedups, we can speculate that there still is non-exploited parallelism and even higher speedups are possible with larger number of processors. This high performance behavior is due to the parallel processing of multiple input packets. Since as many input packets as the number processors can be processed concurrently, FMR application scales well even with large number of processors.

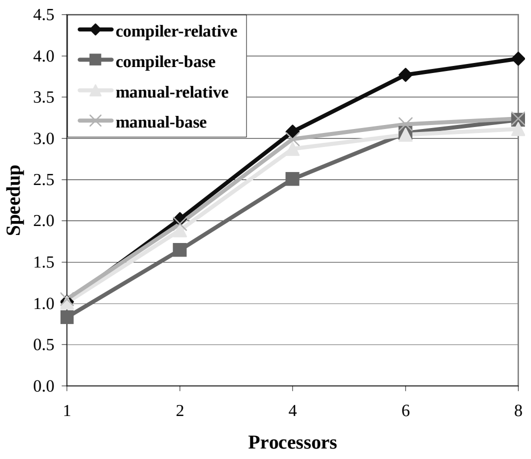
For the GSM application, the speedup scales well up to 8 processors. Unlike MAD and FMR, in GSM, the parallel execution is realized, not by the parallel processing of different input frames with each other; but, in fact, by the parallel processing of a single



(a) The speedup of the MAD application.



(b) The speedup of the FMR application.



(c) The speedup of the GSM application.

Figure 8.1: The speedups of the benchmark applications.

frame. This sets a limitation on the available parallelism and results in a relative-speedup of 4 with 8 processors for the compiler-optimized version.

The fact that all three applications scale well proves that the applications benefit from the proposed code transformations (Chapter 5), whether the transformations are applied manually by a programmer or automatically by a compiler.

Comparison of the Manually and Compiler Optimized Versions

The differences between the speedups of the manually and compiler optimized versions are little for the three benchmark applications.

For the MAD application, the speedups of the manually and compiler optimized versions completely match.

For the FMR application, the speedups of the manually-optimized version are higher than that of the compiler-optimized version. This difference is caused by the ability of the arm-elf-gcc compiler to better optimize the compiler-optimized version than both the manual-optimized version and the baseline. We noticed that this behavior, which will be demonstrated with experimental results in the following section, expands the parallel portion of the FMR application in the manually-optimized version and results in better speedups.

For the GSM application, the base speedups for both the compiler-optimized and the manual-optimized versions are close. In fact, with 8 processors the relative-speedup is flattened at 3.1 with the manual-optimized version; however, with the compiler-optimized version there still exists non-exploited parallelism, i.e. the speedup can potentially increase. This demonstrates that the MOC is more successful in extracting parallelism, than the programmer. We can speculate that buffer renaming code transformation is the main reason of this behavior because it requires a complete comparison of the buffer sections for every task pair in the control program and therefore is not easily applied by a programmer.

The fact that the speedups of the compiler-optimized versions are close to that of the manually-optimized versions (close for MAD and FMR, and even higher for GSM) proves that the implemented Sarek-compiler is able to apply the necessary code transformations that the applications benefit from.

Comparison of the Base and Relative Speedups

For the MAD application, the difference between the base and relative speedups is caused by the overheads introduced by the task selection and the code transformations, which will be discussed in the remainder of this section.

For the FMR application, the base speedup is higher than the relative speedup for the compiler-optimized version. This is contrary to what one expects because the relative speedup reflects overheads that do not exist in the base speedup. We conjecture that this behavior is also the result of the arm-elf-gcc optimizations, which will be discussed in the following section.

For the GSM application, the difference between the base and the relative speedups is mainly because of the overhead introduced by the task selection.

The Impact of the Overheads

The impact of the overheads on the execution cycles will be investigated in the following section. We report on the offsets that the overheads introduce on the speedup results.

The MAD application exhibits the overheads introduced by the task selection and parameter deaggregation code transformation. These cause a base-speedup of 0.6 with one processor[3].

The FMR application exhibits the overheads introduced by the task selection. However, again the `O2` optimization level of arm-elf-gcc, eliminates the effect of the task selection and code transformations overheads in compiler-optimized version and causes

[3]Again, buffer privatization and replication overheads are hidden by the MemoryProc.

a base-speedup of 1.1 with one processor.

The GSM application exhibits the overheads introduced by the task selection, which causes 0.8 base-speedup with one processor for the compiler-optimized version.

8.4.2 Overhead Experiments

In a compiler-optimized application, three main sources of overheads have impact on the total execution cycles. First, the code transformations introduce overheads, by increasing the total number of instructions (buffer privatization and buffer replication) and the number of input and output arguments (parameter deaggregation and buffer renaming). Second, because dispatching a task takes longer than a function call due to the delays in the CP and the task dispatcher, renewing the task selection with finer-grain tasks introduces overheads, due to the increased number of task calls in the control program. Since more task calls also result in more input and output arguments, a task selection with finer-grain tasks yields even more overheads caused by the increased number of URF accesses for reading/writing the task arguments from/to URF. It is important to note that the overhead of task selection is dependent on the MLCA parameters and may have significantly different effects on the execution cycles in a real MLCA instance. Third, since the task functions need to be modified according to the outcome of the code transformations, the MOC includes a code generator, as explained in Section A.2. Because transforming the intermediate representation of the MOC back to the C language, does not always yield the original input code, possible extra instructions may cause overheads in the generated task functions, even though arm-elf-gcc optimizations are expected to clean most of them.

Among the described sources of overheads for the compiler-optimized versions, the manually-optimized versions include the task selection and the code transformation overheads. On the other hand, since the baseline versions contain a single task, no compiler generated task functions and no code transformation, they do not reflect any overhead.

In order to evaluate the effect of different overheads on the execution of the applications, we prepare two more versions of each application, in addition to baseline (B), manually-optimized (MO) and compiler-optimized (CO) versions.

The **OV1** is a version of the application where the tasks and the control program are selected and are the same as in the compiler-optimized version; on the other hand, this control program is not optimized via any code transformation and is not processed by the MOC. In that sense, an OV1 version does experience the task selection overheads, but not the overheads introduced by the code transformations and the compiler generated task functions.

The **OV2** is a version of the application that the OV1 version is given to the MOC as input; however, the code transformations are disabled and, thus, not applied by the MOC. On the other hand, the MOC produces the task functions as it would if the code transformations were applied, even though the task functions are not modified. Consequently, the OV2 versions exhibit the overheads of the task selection and the compiler generated task functions, but not the code transformation overheads.

We compile all 5 different versions of each benchmark application with `O2` optimization level of arm-elf-gcc and run them with one ARM processor and no MemoryProc. This eliminates the effect of MemoryProc on the speedup results, discussed in previous section, and enables a complete comparison of the different overheads. Figure 8.2 depicts the execution cycles of the different versions of MAD, FMR and GSM applications, normalized with respect to the baseline of the corresponding application.

For the MAD application, the 54.6% of difference between the baseline and the OV1 version represents the overhead of the task selection, which is high due to the fine-grain tasks that exist in the application. Furthermore, a code transformation applied during the task selection process, which duplicates a portion of a large task to enable the early computation of some loop-carried task arguments, also contributes to the task

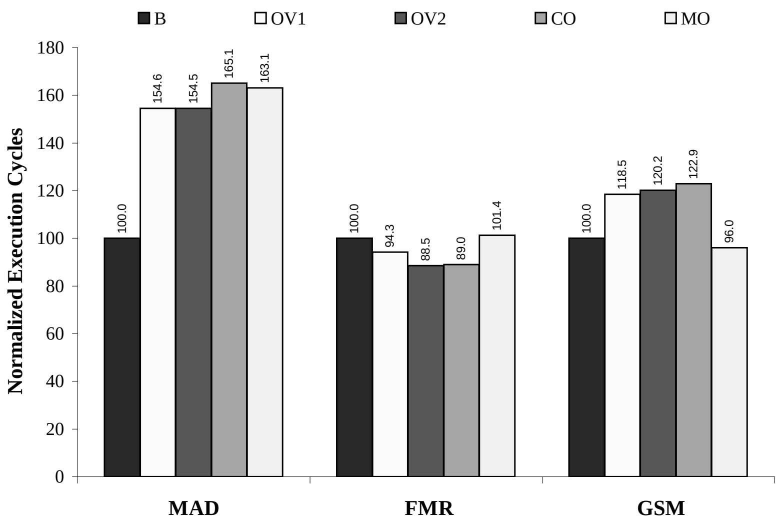


Figure 8.2: Single processor execution cycles for different versions of the benchmark applications.

selection overhead with a ratio of 10% with respect to the baseline[4]. The fact that the execution cycles of the OV1 and the OV2 versions are the same proves that the compiler generated task functions do not cause significant overheads. Moreover, the 10.6% difference between the OV2 and the compiler-optimized version represents the overhead of the code transformations. As a result, the total of 65.1% overhead introduced by the task selection and the code transformations causes the 3.9 relative speedup to drop to 2.4 base-speedup for the MAD application, in Figure 8.1.

For the FMR application, the expected pattern of increasing execution cycles, from the baseline towards the compiler-optimized version, is not seen because of the optimizations applied by arm-elf-gcc. In fact, we conjecture that the arm-elf-gcc optimizes the application's code significantly better after the task selection is performed compared to the baseline. Moreover, the compiler generated task functions results in even better optimization with arm-elf-gcc. In fact, arm-elf-gcc optimizations cause a 5.7% of drop in the execution cycles with OV1 version and another 5.8% of drop with the OV2 version, with

[4]This is the only transformation, other than the Sarek code transformations, applied to any of the three applications.

respect to the baseline. The 0.5% increase between the OV2 and the compiler-optimized version is caused by the overheads of the code transformations.

In order to prove our conjecture, i.e. the effect of arm-elf-gcc optimizations on the execution of FMR, we compiled all 5 versions of FMR with no arm-elf-gcc optimization and repeated the experiments. With `O0` optimization level of arm-elf-gcc, unlike `O2`, the OV1 version took 5.4% more cycles to complete than the baseline, OV2 resulted in 35.8% of slow down compared to OV1 version (caused by the overhead of compiler generated code), and a 13% overhead is introduced by the code transformations to the compiler-optimized version. The reason of the significant difference in terms of execution cycles, between the `O2` and `O0` optimization levels of arm-elf-gcc, is due to the approach followed during task selection. With this approach, constant scalars, passed as input arguments to function calls, are propagated to the function bodies, when such functions are transformed to tasks. This manual inter-procedural constant propagation enables intra-procedural constant propagation, which can not be performed in the baseline, because arm-elf-gcc can not successfully apply the inter-procedural constant propagation. Furthermore, when applied to the compiler generated task functions, the `O2` optimization level is even more successful, speculating that the MOC opens up more opportunities for the arm-elf-gcc optimizations. As a result, the total of 11% decrease in the execution cycles of the compiler-optimized version compared to baseline, causes the base-speedup of the FMR application to be higher than its relative speedup, in Figure 8.1.

For the GSM application, compared its baseline, the task selection overhead causes an 18.5% slow-down in the OV1 version. Furthermore, the compiler-generated task functions cause an additional 1.7% overhead in the OV2 version and the code transformations produce another additional 2.7% overhead in the compiler-optimized version. As a result, a total of 22.9% of slow-down is experienced by the compiler-optimized version compared to the baseline, which reduces the relative-speedup of 4 to a base-speedup of 3.2 with 8 ARM processors (in Figure 8.1). The fact that the manual-optimized version is 4%

faster than the baseline (in Figure 8.2), is caused by some unused functionalities of the GSM application, which are omitted by the programmer. This causes a bias of 0.1 in the speedup curves of manual-optimized version, in Figure 8.1, which does not affect the slope of the speedup curves, i.e. extracted parallelism. In fact, when executed with 8 ARM processors, the base-speedup of the compiler-optimized version overcomes this bias, by extracting more parallelism, and exceeds the base-speedup of the manual-optimized version.

8.4.3 Code Transformation Experiments

In order to test the effectiveness of the code transformations applied by the MOC, we experiment with 5 different versions of each application, incrementally enabling each code transformation in the MOC. In these experiments, the complete buffer section, structure fields, allocation and deallocation pragmas are obtained through code inspection and manually provided to the MOC, in order to obtain the maximum performance out of the code transformations

The **OPT0** is the version in which all the code transformations are disabled and, hence, is the same as OV2 version described earlier.

The **OPT1** is the version to which only parameter deaggregation is applied.

The **OPT2** includes parameter deaggregation and buffer privatization code transformations.

The **OPT3** is applied parameter deaggregation, buffer privatization and buffer replication code transformations.

The **OPT4** is the version to which all the code transformations (parameter deaggregation, buffer privatization, buffer replication and buffer renaming) are applied, and, therefore, is the same as the compiler-optimized version described previously.

In this experiment, the code transformations are incrementally applied because they are effective on the outcome of each other. Parameter deaggregation opens up more opportunities for the buffer transformations, by extracting buffers inside structures. Buffer privatization, by resolving memory false dependences effectively, reduces the need for buffer replication and, therefore, the overheads. Buffer privatization and buffer replication, by resolving memory false dependences, create more parallel tasks and open up more opportunities for buffer renaming. In addition, as our task function bodies are minimal in size, code hoisting did not improve the performance of the applications. Therefore it is not included in our evaluation.

The codes of the applications are compiled with `O2` optimization level of arm-elf-gcc and the control program is run with 8 ARM processors and a MemoryProc. Figure 8.3 shows the execution cycles of each version described above, normalized with respect to the `OPT0` version.

Parameter deaggregation has no effect on the execution cycles of FMR, because FMR does not use any structures. On the other hand, it is seen that parameter deaggregation speeds up MAD by 9.3% and GSM by 3%. These improvements are caused by the deaggregated two main structures of the MAD and GSM applications. However, in GSM, the deaggregated structure consists mainly of buffers. As it is explained in Section 6.3, the structure fields of type buffer are marked both as input and output arguments to the tasks, during the parameter deaggregation, in order not to violate the false memory dependences. As a result, the GSM application benefits little from parameter deaggregation by itself because false dependences of the deaggregated buffers are not resolved via buffer code transformations, i.e. buffer privatization, replication and renaming. On the other hand, the deaggregated structure of MAD consists mainly of scalars. These scalars, when deaggregated, introduce no false dependences among tasks, due to the renaming mechanism of the MLCA. As a result, in MAD, some parallelism is obtained among the tasks accessing these scalars. However, since the deaggregated structures of

GSM and MAD contain buffers, buffer code transformations are necessary to get the most parallelism out of the parameter deaggregation.

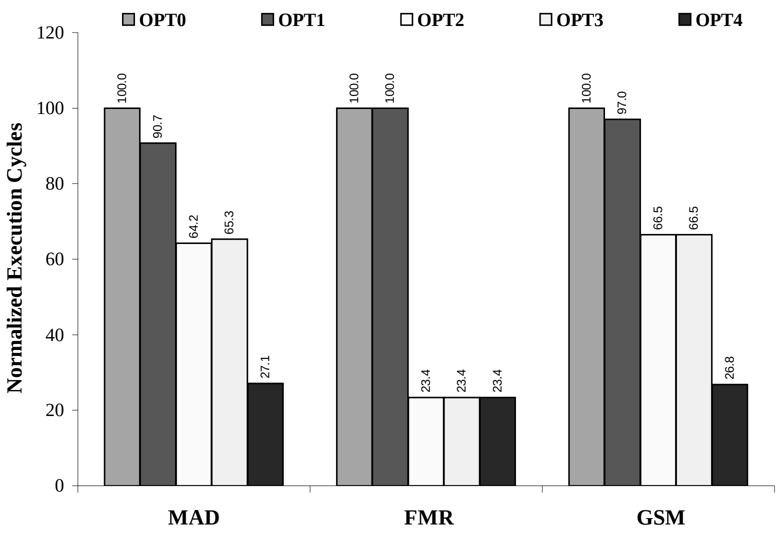


Figure 8.3: The effect of the code transformations on the total execution cycles.

Buffer privatization reduces the execution cycles 26.5% in MAD, 76.6% in FMR and 30.5% in GSM after the parameter deaggregation is applied. These improvements are due to 18 buffers privatized in MAD, 21 buffers privatized in FMR and 15 buffers privatized in GSM.

Buffer replication is not applied in FMR and GSM, because all the buffer replication candidate task pairs are dependent to each other with true dependences. On the other hand, buffer replication causes a 1.1% of drop in MAD performance after the privatization. This is due to the optimizations applied by the arm-elf-gcc which reduces the execution cycles of the replication target tasks. Since the overheads of allocating, initializing and deallocating the replica is not affected by the arm-elf-gcc optimizations, buffer replication decreases the performance of the MAD application. However, for MAD with no arm-elf-gcc optimizations and during the development of the MOC, we noticed that buffer replication is beneficial. Buffer replication is especially effective for applications which contain many loop-carried buffer true dependences and, hence, buffer privatization is ineffective. Since, all three of our benchmark applications greatly benefit from buffer privatization, not much improvement in performance is left for the buffer replication.

Buffer renaming improves the performance of MAD by 38.2% and the performance of GSM by 39.7%, after the buffer replication. These improvements in GSM and in MAD are due to a large number of tasks that access different sections of a buffer. When the synchronization false dependences among these tasks are resolved with buffer renaming, these tasks can execute in parallel, improving the performance.

In conclusion, when provided with complete buffer sections and structure field definition/uses information, the MOC is successful in extracting parallelism via the proposed code transformations discussed in Chapter 5. This success results in equally good performance as manual-optimized versions in MAD and FMR applications and a better performance in GSM.

Furthermore, most of the overheads which reduce the performance of the compiler-optimized applications are, in fact, caused by the task selection, whereas the affect of the compiler generated code is insignificant with `O2` optimization level of arm-elf-gcc, and the code-transformations introduce little overheads (2.7% with GSM, 0.5% with FMR and 10.6% with MAD).

Finally, parameter deaggregation not only opens up opportunities for buffer code transformations, but also improves the performance, when deaggregated structures include scalars (9.3% in MAD). Buffer privatization is very effective in extracting parallelism from the multimedia applications that involves frame/packet processing (26.5% improvement in MAD, 76.6% in FMR and 30.5% in GSM). Buffer renaming, on the other hand, is effective in applications that involves accesses to different sections of buffers. In fact, the speedup is almost tripled with buffer renaming in MAD and GSM.

8.5 Performance with the ORC C-Compiler

In order to evaluate to the ability of today's compilers (specifically ORC [7]) in generating the pragmas, we use four versions of each application.

The **OPT0** is the version to which no code transformation is applied.

The **PRAGMA1** is the version in which only the buffer and structure allocation and deallocations are provided with manually defined pragmas, in order to enable code transformations. However, the buffer sections and structure fields definition/use pragmas are generated by the C-compiler and the code transformations are applied by the MLCA compiler according to these provided pragmas.

The **PRAGMA2** is the version in which, in addition to allocation and deallocation pragmas, buffer region pragmas are provided manually in a way to fix the problems of the ORC array section analysis, discussed in Chapter A. In other words, rather than providing all the buffer regions with pragmas, only the regions of the buffers inside the structures are provided. In addition, since ORC array section analysis is flow-insensitive, buffer section `NoUses` pragmas are provided, in case a buffer section is not used, but is defined.

The **PRAGMA3** is the version in which all the buffer section and structure fields definition/use pragmas are provided manually and is the same as the compiler-optimized version described in the previous section.

We compile each version of each benchmark application, with `O2` optimization level of the arm-elf-gcc and run them with the MLCA instance of Table 8.1, 8 ARM PUs and a MemoryProc. Figure 8.4 depicts the execution cycles of each version of each benchmark application, normalized with respect to the OPT0 version.

For the MAD application, with PRAGMA1 version, a speedup of 9.3% is experienced compared to the OPT0 version. This speedup is due to the correct structure fields data-

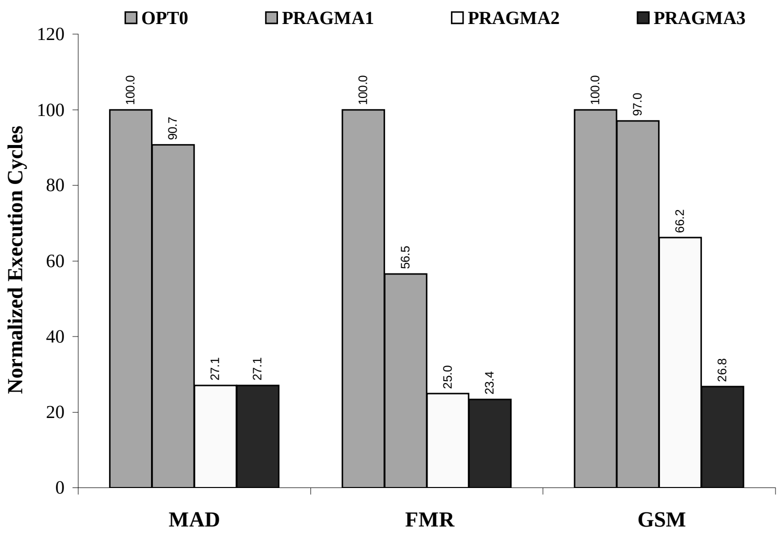


Figure 8.4: The effect of user pragmas on the total execution cycles.

flow analysis performed by the C-compiler. As a result, the parameter deaggregation code transformation is applied completely by the MLCA compiler, resulting the same performance improvement as in the OPT1 version (in which only parameter deaggregation is applied) presented previously. On the other hand, since the majority of the buffers are stored inside structures and excessive pointer arithmetic is used to access the buffers of the MAD application, the array section analysis of the C-compiler is unable to provide the MLCA-compiler with the buffer sections. Consequently, buffer code transformations, i.e. buffer privatization, buffer replication and buffer renaming could not be applied by the MLCA compiler. However, when the buffer regions are provided manually to the MLCA compiler, in the PRAGMA3 version, all the buffer code transformations are successfully applied by the MLCA compiler. Therefore, the full speedup of the application, i.e. the speedup of the application when all the pragmas are provided manually, is obtained, resulting in normalized execution cycles of 27.1%. Since in the MAD application almost all the buffers used are stored in structures, PRAGMA3 version (in which all the buffer regions are provided) is the same as the PRAGMA2 version (in which regions of buffers inside structures are provided).

The FMR application does not include any structure, and thus, the structure fields data-flow analysis is not performed by the C-compiler. On the other hand, since each buffer is accessed in for-loop's and only inside the task function bodies, array section analysis of the C-compiler is successful in predicting the accessed regions. However, the flow-insensitive array-section analysis creates extra memory true dependences and prevents buffer code transformations. Consequently, a 43.5% improvement is obtained with respect to the OPT0 version. When, the flow insensitive analysis results are fixed with `NoUses` pragmas in the PRAGMA2 version, an additional improvement of 31.5% is obtained. The 1.6% of difference between the PRAGMA2 and the PRAGMA3 version is caused by a buffer section predicted conservatively, which prevents buffer privatization for the buffer that this section belongs to.

The GSM application experiences similar performance improvements as the MAD application. When no pragmas are provided manually, the structure field data-flow analysis of the C-compiler, provides the required pragmas for the parameter deaggregation transformation and a 3% improvement is obtained as a result. When the buffer sections are provided manually for the buffers of the structures and `NoUses` pragmas are provided for the buffer sections not used inside the tasks, a normalized execution cycles of 66.2% is obtained with PRAGMA2 version. The fact that, even with the described buffer pragmas, the performance is significantly lower than the PRAGMA3 version, is caused by a large number of buffers for which the sections are predicted conservatively by the C-compiler (ORC). This is due to, again, the excessive pointer arithmetic usage in GSM.

In conclusion, for applications that do not involve pointer arithmetic and use loops to access buffers, such as FMR, the array section analysis of the C-Compiler is successful in predicting the buffer sections. On the other hand, the lack of flow-sensitive analysis and the fact that buffers inside the structures are ignored by the C-Compiler during the section analysis, limits the success of the MLCA compiler in applying the code transformations. Alias analysis, flow sensitiveness, analyses of the buffers inside structures and inter-procedural array-section propagation are needed in the C-compiler to make its array-section analysis successful enough for the MOC to apply the Sarek code transformations. Implementing these improvements is outside the scope of this thesis.

Furthermore, unlike the array-section analysis of the C-compiler, the implemented structure fields inter-procedural data-flow analysis is successful in providing the MLCA compiler with the definition and use pragmas for the structure fields. As a result, parameter deaggregation code transformation is successfully applied for the MAD and GSM applications, opening up more opportunities for buffer code transformations.

Chapter 9

Related Work

There exists a number of SoC systems that use multiple processing units for multimedia and other applications [1, 2, 8, 13]. Daytona's scalable DSP architecture [13] features multiple processors with a split-transaction bus for communication and cached semaphores for synchronization. The picoChip [8] is a cascadable reconfigurable architecture of array processors intended for 3G wireless communications. Cradle's Technologies's *3SOC* is a shared-address space multi-processor SoC. It consists of a number of processor clusters that are connected by two levels of buses [1]. The system provides 32 semaphore registers for synchronization, which must be explicitly used in a parallel program. All these systems require their users to explicitly express applications as parallel programs. In contrast, the MLCA is programmed in a semi-sequential programming model.

Our code transformations build on a number of well-known compiler analyses and optimizations, including privatization, section analysis, dependence analysis and hoisting.

Privatization [12] is an optimization technique applied by parallelizing compilers to improve loop-level parallelism in programs. Parallel Computing Forum (PCF) Fortran [14], IBM parallel Fortran [26] and OpenMP [6] include private declarations for scalars and arrays in the context of loops, which enable the programmer to declare an array or a scalar private to the iterations of a loop.

Tu and Padua [27] propose a technique to automatically apply array privatization. Their algorithm use a data flow analysis to find arrays that are used in an iteration of a loop but are not exposed to definitions outside the iteration. Such arrays are marked as privatizable. Then, each privatizable array is tested for profitability. If different iterations of a loop access the same set of locations in a privatizable array, this array is considered as profitable for privatization. Finally, each array to be privatized is tested for liveness after the loop. If array data is used after the loop, the used locations are copied to the privatized array after the loop. If array data is not used after the loop, no data is copied to the privatized array.

Our buffer privatization transformation bears similarities to the array privatization approach of Tu and Padua. First, similar to our buffer section data flow analysis, they perform data flow analysis to detect the array references that are not exposed to definitions outside a loop iteration. Second, they conclude that privatizing an array is profitable, if iterations of a loop access the same locations of the array. We perform the same efficiency check in our buffer privatization algorithm by comparing accessed regions of each task.

However, our buffer privatization transformation is also different from array privatization, mainly due to the granularity of the targeted parallelism. Array privatization aims for parallelism among loop iterations, whereas buffer privatization aims for task-level parallelism among task calls in a control program. This results in three major differences between the two transformations. First, for detecting privatizable arrays, array privatization performs data flow analysis on the array element accesses in a loop body. In contrast, buffer privatization performs data flow analysis on the array section accesses by tasks in a control program. Second, array privatization privatizes arrays for whole loop iterations, whereas buffer privatization privatizes buffers for collection of tasks. This enables buffer privatization to extract more parallelism, since each buffer may be privatized several times in a single loop iteration, resulting body-level parallelism. Third, in case a buffer is accessed beyond a loop, buffer privatization conservatively marks the buffer as not privatizable for the iterations of this loop. Array privatization, on the other hand, uses live value analysis to detect the contents of the privatized array after the loop is executed and performs the privatization.

The buffer section data flow analysis used in the algorithms of our code transformations is similar to the symbolic array dataflow analysis technique proposed by Li and Lee [16]. They compute defined, used and killed regions of arrays in each procedure and propagate these array regions along the call graph to enable coarse grain parallelism optimizations, such as array privatization. They use similar array section operations, such as intersection, union and difference, to the ones used in our buffer section data flow analysis. However, since our buffer section data flow analysis is only applied to control programs, we use a limited (only intra-procedural) version of the array dataflow analysis. Furthermore, they use guarded array regions which associate context with array region accesses. We believe that, if the C-Compiler of the MOC incorporates the symbolic array dataflow analysis with guarded array regions, our buffer section analysis can easily be improved to process these guarded regions. This will increase the effectiveness of our code transformations.

We use simple representations of array sections in Sarek pragmas and buffer section analysis algorithm. Balasundaram and Kennedy [10] use similar array section representations, referred to as *simple sections*, to enhance the task level parallelism in programs. However, they concentrate on detecting parallel tasks, and on pipelining tasks. We use array sections to create parallel programs from sequential programs. More complex representations of array sections are proposed by Hoflinger [24] to be used in array section analyses.

Our code transformations rely on the characteristics of multimedia applications. Fritts et. al [15] gives an overview of multimedia applications. Lee et. al [22] propose MediaBench [5] as benchmark tools to represent the class of multimedia applications. We use

GSM application, which belongs to MediaBench, in our experiments.

Chapter 10

Conclusion and Future Work

10.1 Conclusion

The MLCA is a SoC architecture that incorporates a control processor (CP), multiple processing units (PUs), a universal register file (URF) and a shared memory. It is intended to support a convenient programming model for the multimedia applications, and provide high performance. The CP dispatches coarse grain computation units, called tasks, to the PUs. Each task reads from and/or writes to the URF its input and output arguments respectively. The CP keeps track of the URF dependences between tasks and synchronizes them accordingly. It also renames the URF registers to resolve false dependences between tasks. The MLCA programming model consists of a control program that represents the control flow of the tasks and their arguments, and task functions. Control programs are written in a high level programming language called *Sarek*, whereas the task functions can be written in a regular programming language, such as C.

Despite the benefits of the MLCA programming model, the naive expressions of a multimedia application as a control program and task bodies may cause incorrect execution and/or poor performance. This is often caused by the usage of the pointers (in the URF) to data in shared memory, which renders the synchronization and renaming mechanisms of the MLCA ineffective. Since hardware can only rename the URF registers, the false dependences in the shared memory are not resolved, which reduces performance. Thus, compiler support is required for the MLCA in order to handle these correctness and performance issues.

In this thesis, we described the MLCA Optimizing Compiler, designed to facilitate the process of porting applications to the MLCA programming model. It handles the correctness and performance issues by applying four code transformations collectively referred to as the Sarek code transformations. First, parameter deaggregation moves scalar data inside shared memory structures to the URF, which enables the hardware to resolve the false dependences among these scalars. Second, buffer privatization creates a private copy of a buffer in shared memory for a collection of tasks, which resolves the false dependences among these tasks, caused by the accesses to that buffer. Third, buffer replication generates a copy of the buffer to be read in a single task, which resolves the memory false dependence(s) between this task and the subsequent tasks that write to that buffer. Finally, buffer renaming prevents incorrect execution caused by violated data dependences, by reorganizing the arguments according to the data dependences caused by the shared memory accesses.

We have implemented a prototype implementation of the MLCA Optimizing Compiler using the ORC open source compiler as the infrastructure. The MLCA Optimizing Compiler consists of two main compilers: the C and the Sarek compilers, which respectively analyze the task bodies and optimizes the control program. The C compiler inserts the analyses results of the task functions into the code of the tasks in the form of pragma statements which are later extracted and processed by the Sarek Compiler. The compiler analyses performed by the C compiler are inter-procedural array section analysis and the inter-procedural data flow analysis. In order to address some of the limitations introduced by the ORC, an API is also provided to allow programmers to provide high-level data usage information into the application code. We believe that such information can easily be obtained by just inspecting the code of an application by a programmer with reasonable understanding of the application.

In order to assess the benefits of our code transformations, we ported three multimedia applications (MAD, FMR and GSM) to the MLCA programming model and reported on their performance using a functional simulator of the MLCA. When provided with perfect analyses results via the pragma API, scaling speedups are obtained with all three applications: 3.9, 4.8 and 4.0 in MAD, FMR and GSM, respectively, with 8 processors. These speedups are comparable to the speedups of the hand-ported versions of the same applications: 3.9, 6.8 and 3.1, respectively, with 8 processors. Our experiments showed that the overheads of the code transformations are negligible. We also evaluated the individual contributions of each code transformation to the overall speedup. The results showed that all the code transformations contribute to the performance increase, except the buffer replication code transformation which we believe is effective in different circumstances than our applications. More specifically, the parameter deaggregation improved the performance up to 9.3% in MAD and 3.0% in GSM, which both contain structures. Buffer privatization resulted in 26.5% and 30.5%, and buffer renaming resulted in 38.2% and 39.7% performance increases in MAD and GSM applications. Buffer renaming did not contribute to the speedup in FMR application because buffer privatization exploits all the available performance with a performance increase of 76.6%.

We also experimented with task analyses results provided by the ORC compiler. These experiments showed that the inter-procedural data-flow analyzer that we implemented with the ORC is successful in producing the necessary pragmas to perform the parameter deaggregation code transformation. Nevertheless, the inter-procedural array section analysis is too conservative and without the contribution of the programmer in providing accessed buffer sections inside tasks, it is unable to provide the Sarek compiler with satisfactory results. On the other hand, in applications that include simple accesses to buffers without any pointer usage, such as in the FMR application, the array section analysis may provide the information required by the Sarek compiler, which results in increased speedup. In fact, the performance increased by 43.5% in the FMR application without any user pragmas.

10.2 Future Work

There are a number of future work directions that either address some of the limitations of this work or extend it.

The inter-procedural data flow analysis and the inter-procedural array section analysis performed by the C-Compiler (i.e. inter-procedural analyzer of ORC) are conservative because they are flow-insensitive. This is a limitation introduced by the ORC compiler and the inter procedural analysis phase (IPA) of ORC can be enhanced to overcome this limitation. Nevertheless, we should note that, in our experiments, the implemented flow-insensitive inter procedural data flow analysis for structure fields produced perfect results and, thus, the parameter deaggregation transformation could be applied even in the absence of the user pragmas.

The inter-procedural array section analysis of the ORC originally does not consider buffers that are inside structures or that are accessed using pointer arithmetic. Also, when the address of a buffer is passed to a function, no array section information is generated for the buffer in the function. These limitations reduce the accuracy of the array section analysis significantly. Thus, the intra-procedural analyzer (IPL) phase of the ORC compiler can be enhanced.

The current design of the code transformations and implementation of the Sarek and C compilers assume no aliasing of task arguments. This is a realistic assumption for the Sarek compiler because multimedia applications usually involve no aliasing among pointers passed to the tasks as arguments. In fact, a buffer or a structure created in the beginning of a program is accessed in each task with its start address, which does not change throughout the execution of the program. Although we have designed pragmas to handle simple cases of aliasing in the control programs, our benchmark applications did not include these cases. Nevertheless, the code transformations can be altered to handle aliasing among the task arguments to improve effectiveness. The aliasing in the C-Compiler, on the other hand, affects the produced analysis results. The implemented inter procedural data flow analyzer for the structure fields takes into account aliasing among the structure pointers, thus, it does not require any enhancements. However, because the original array section analyzer of the ORC does not consider pointers, any new functionality to analyze buffer pointers would require alias analysis.

Our definition and usage of array sections is context insensitive. This may limit parallelism in some cases. Thus, a possible extension of our work is to use guarded regions [16] proposed by Gu et al. that allow a region to be associated with context. Also, symbolic lower and upper bounds for array sections may improve the effectiveness of the code transformations.

The work described in this thesis opens up opportunities for different research topics related to the MLCA. As the presented compile-time code transformations enhance the performance of the ported applications, solutions to the remaining issues of porting applications and reducing the resource usage can be focused. A promising research area would be solving the task selection problem that was mentioned in different contexts throughout the thesis. Furthermore, task scheduling for further improving the performance and/or reducing power consumption is an interesting problem. Several architectural design questions, such as the structure of the URF, PU interconnection or memory model in the MLCA are also open for investigation.

Appendix A

Compiler Implementation

This chapter presents an overview of the MLCA Optimizing Compiler (MOC) implementation. Section A.1 briefly describes the selected infrastructure, i.e. ORC. Section A.2 describes the implementation of the MLCA Optimization Compiler, together with the C and Sarek sub-compilers. The compilation phases and the major modules are presented to describe the Sarek-Compiler. In addition, the modifications performed on the ORC in order to adopt the necessary compiler analyses are briefly discussed for the C-compiler.

A.1 The ORC Compiler

In this section, we describe the compiler infrastructure selected for implementing the MLCA Optimizing Compiler. We justify our decision and briefly present the features of the infrastructure.

We have selected Open Research Compiler (ORC) [7] as the infrastructure for both C and Sarek compilers, because of several reasons:

1. It is effort and time saving to build the MLCA compiler prototype as a part of an existing compiler in order to benefit from the intermediate representation (IR) tools and data structures.

2. ORC is an open source compiler infrastructure.

3. ORC is designed for robustness, performance, and flexibility. Moreover, there is an increasing interest towards ORC among compiler research community.

4. ORC is based on the MIPSpro compiler of SGI and went through several re-designs and enhancements.

5. ORC includes C/C++/Fortran front ends.

6. ORC has tools for most of the analyses needed for the MLCA Optimizing Compiler, including array-section analysis and inter-procedural data-flow analysis.

7. ORC's source code is well structured and tools for manipulating IR are satisfactory.

ORC has the major components of C/C++/Fortran front-ends, inter-procedural analyses and optimizations, loop-nest optimizations, scalar optimizations and code generation. It is designed for Linux platform and targets IA64 architectures. Its intermediate representation is called *Whirl*. Whirl is AST based and provides communication between gfec (C front-end), gfecc (C++ front-end), F90 (Fortran front-end), IPL (intra-procedural analyzer), IPA (inter-procedural optimizer), LNO (loop-nest optimizer), WOPT (global optimizer), whirl2c (Whirl-to-C converter), whirl2f (Whirl-to-Fortran converter) and CG (code-generator) phases.

Multi-level lowering is the most significant property of the Whirl. Whirl consists of four different levels: very high, high, medium and low. Higher levels are more structured than lower ones, however; lower levels contain more information and is closer to assembly. Whirl is continuously lowered during compilation and low Whirl is finally transformed to assembly during the CG phase. Figure A.1 depicts the lowering of Whirl between different phases of ORC.

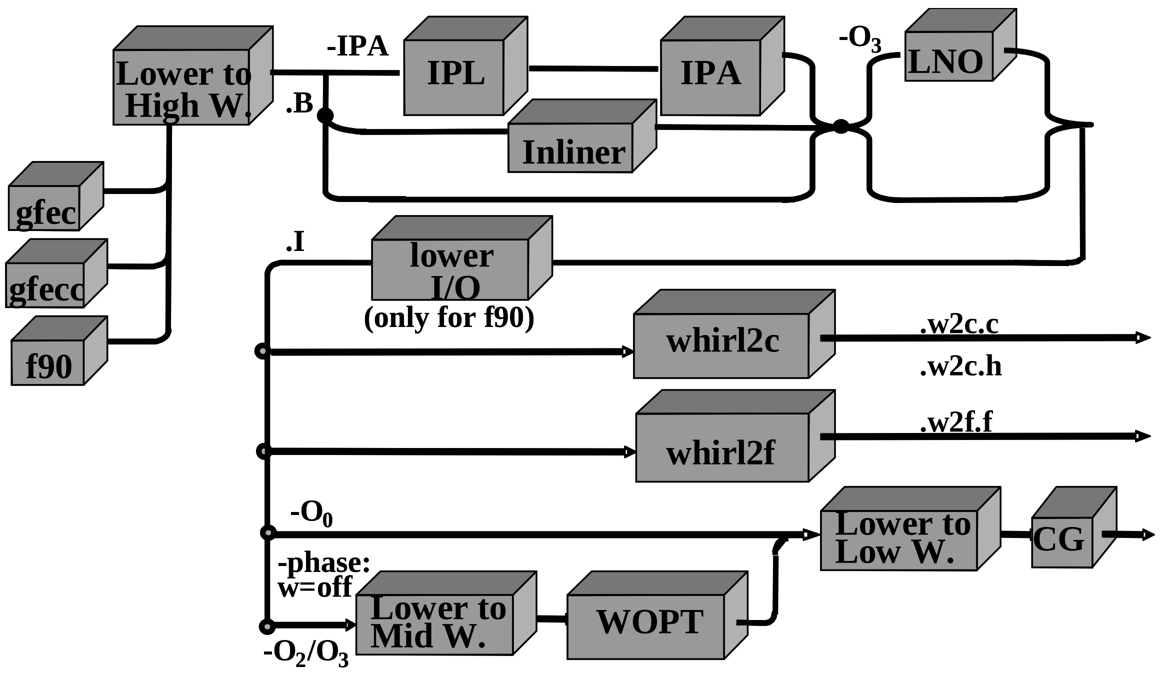


Figure A.1: ORC phases and Whirl lowering [7].

A.2 The MOC Implementation

The overall architecture of the MOC is depicted in Figure A.2.

The C-compiler and the Sarek-compiler are both implemented with ORC. For ease of implementation and faster compilation, the communication between the Sarek-Compiler and the C-Compiler, via the Sarek pragmas, is realized with intermediate object files. In other words, the C-compiler generates the object files including the Sarek pragmas for the task functions. The Sarek-compiler takes these object files, opens the Whirl tree of the task functions and the symbol table, extracts the Sarek pragmas from the Whirl tree, and optimizes the input control program using the analyses results contained in the Sarek pragmas. The described implementation is possible due the fact that early phases of ORC include the necessary compiler analyses (such as inter-procedural data-flow and array-section analyses), and its late phases are suitable for implementing the Sarek-compiler.

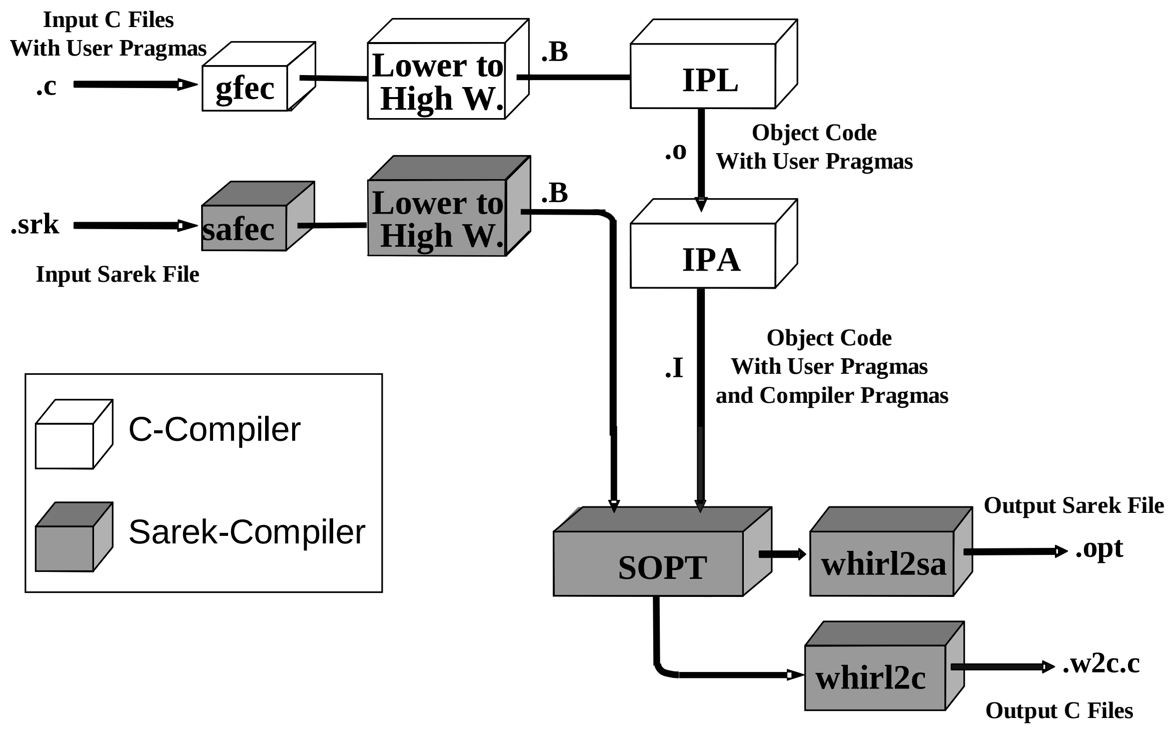


Figure A.2: The implementation of the MLCA Optimizing Compiler.

In the remainder of this section, the implementations of the Sarek-Compiler and the C-Compiler are described.

A.3 The Sarek-Compiler Architecture

In order to optimize control programs, a Sarek front-end, a Sarek optimizer and a Sarek code generator are implemented. In addition, in order to generate the modified task functions after the Sarek code transformations, a C code generator is also incorporated into the Sarek-Compiler.

A Sarek front-end is designed for the Sarek grammar, based on the ORC C front-end and named as *safec*. For implementing the Sarek code transformations, the global optimizer (WOPT) of ORC is used, where the control-flow, data-flow and regular scalar transformation tools are available. The new global optimizer that is adopted for Sarek is called the *Sarek Optimizer (SOPT)*. In addition, a Sarek code generator is implemented for transforming Whirl to the Sarek syntax. For this purpose, the whirl2c module of the ORC, which transforms Whirl to C, is modified for Sarek language grammar and the new tool is called *whirl2sa*. Furthermore, the original ORC design is modified to attach whirl2c and whirl2sa at the end of WOPT, because in the original implementation of ORC (Figure A.1), whirl2c and WOPT are in fact independent and alternative tools. In order to run the phases in the appropriate order, the universal driver of ORC is altered.

The compilation starts with parsing the input control program in the safec. The result of the parser is very-high Whirl, which is later lowered at end of safec. The output of safec is a .B file which contains high Whirl. This .B file of the control program is transferred to SOPT. On the other hand, the task functions and the other functions of the application are processed by the C-compiler and object codes for each file containing the pragma annotations are given to the SOPT phase of the Sarek-Compiler. SOPT reads the .B file and the object files of tasks, and processes the Whirl code of the control program, using the annotations extracted from the Whirl code of the task functions. The output of SOPT is the Whirl code of the optimized control program and the modified task functions. These are then transferred to whirl2sa and whirl2c to produce the output Sarek and C codes respectively. The output of whirlsa is the optimized control program in a .opt file, whereas the output of whirl2c is the modified task functions in .w2c.c files.

In the remainder of this section, we will describe the implementation of the Sarek optimizer phase, i.e. SOPT. The major modules implemented and the communication among these modules will be presented.

The SOPT takes Whirl representation of a control program and task functions as input, applies the Sarek code transformations discussed in Chapter 5, and generates the Whirl representation for the optimized control program and the modified task functions.

Support for the Sarek code transformations is added to the original WOPT of ORC with three major modules. The purpose of each module and the communication among

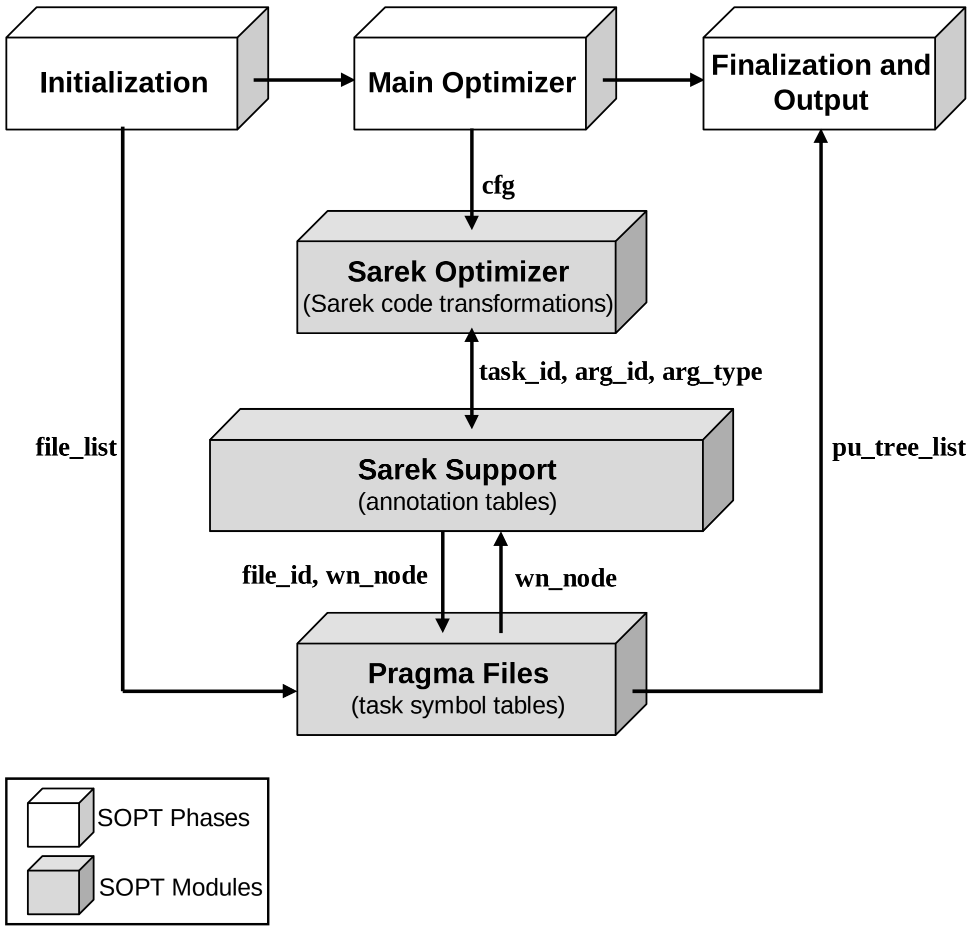


Figure A.3: SOPT phases and modules.

them are depicted in Figure A.3.

Pragma Files Module

Pragma Files Module (PF) consists of classes and tables for managing the files of input task functions. For this purpose, the list of the files is kept in the PF and the necessary tools to open and process the contained Whirl code are implemented.

As a part of initialization in the SOPT, the list of files containing tasks, i.e. `file_list`, is given to PF. Then, each file is opened and the Whirl tree is scanned for Sarek pragmas, in the form of pragma statements. The annotations found (`wn_node`) are sent to the Sarek Support Module for processing.

Since the global optimizer of ORC (WOPT) is originally designed to process only one file at each invocation, it contains functionality to store only one symbol table. However, Sarek optimizer (SOPT) should process multiple files (a Sarek file and multiple files for the task functions) and thus, it requires multiple symbol tables for each input file. For this reason, PF also stores the symbol tables for the task functions[1]. Consequently, if the Sarek Support Module needs the symbol table information for a Whirl tree node, it passes the node, i.e. `wn_node` to the PF with the id of the file that the node belongs to. i.e. `file_id`.

Furthermore, at the end of SOPT, the Whirl code of each task function has to be transformed to C language, reflecting the results of the Sarek code transformations. To achieve this, for each task function file, PF asks the Sarek Support Module for the results of the code transformations, updates the Whirl code accordingly and sends the list of the modified Whirl codes of the task functions (`pu_tree_list`) to whirl2c.

Sarek Support Module

Sarek Support Module (SS) is responsible for storing information about each task of the control program. It can be considered as an extension to original symbol tables of SOPT, for the Sarek code transformations. In fact, SS stores the information transferred to the Sarek-Compiler, from either the C-Compiler or the programmer, via the Sarek pragmas.

SS mostly communicates with the Sarek Optimizer Module. It supplies the Sarek Optimizer Module with the required information in exchange with a unique `task_id`, argument type `arg_type` (input/output) and the argument number `arg_id`. In cases that the symbol table for the task function is required, it communicates with PF. For instance, in order to find out whether a symbol is a field of a structure, type table of the task function that the symbol belongs to is required

[1]The symbol table for the control program is the original symbol table of SOPT.

Sarek Optimizer Module

Sarek Optimizer Module (SO) is the optimizer class for Sarek. It is invoked from the original global optimizer procedure of the SOPT. It takes the necessary information from the SS and optimizes the Whirl code of the control program accordingly.

Inside SO, each task call is represented by a basic block id `bb_id` and each argument is represented by an argument index number `arg_idx`. To communicate with SS, SO contains tables mapping `bb_id` to `task_id` and `arg_idx` to `arg_id` and `arg_type`. In addition to these, several classes are used for transformations, among which the control flow, the scalar data flow, the buffer section data flow, several graphs and the pointer webs are some important ones. The control flow graph `cfg` is obtained from the ORC optimizer.

The algorithms of the Sarek code transformations applied by SO are described in Chapter 6.

A.4 The C-Compiler

The purpose of the C-Compiler is to insert the results of the compiler analyses to Whirl representation of the task functions in the form of Sarek pragmas. The required analyses results help the Sarek-compiler to perform the Sarek code transformations. The inter-procedural data-flow analysis for structure fields and inter-procedural array section analysis are the necessary compiler analyses for the Sarek code transformations. The C-compiler includes functionality to perform these analyses and insert their results to the Whirl code of the task functions.

The IPL and IPA phases of the ORC compiler constitute the C-Compiler for the MLCA Optimizing Compiler. This design decision is based on the fact that these phases are originally responsible of the inter-procedural analyses and optimizations, such as inter-procedural constant propagation, inter-procedural alias analyses and inter-procedural dead function and dead global variable analyses. As a part of these analyses, inter-procedural data-flow analysis for global variables and array-section analysis for global arrays are also a part of the IPL and IPA modules. In fact, IPL performs the intra-procedural analyses (such as scalar data-flow and array sections) and inserts a summary of the results of these analyses to the output object code. On the other hand, IPA takes these summaries and merges them, following the call graph of the application.

In addition to IPL and IPA, the C front-end of ORC, i.e. gfec, is modified to include a pragma parser, in order to process the Sarek pragmas discussed in Section 7.2 and included as a part of the C-Compiler.

Furthermore, the MLCA Optimizing Compiler is implemented in such way that the C-Compiler analyses are optional. In fact, if the programmer does not need any help from the C-compiler in terms of providing the pragmas, the overall C-compiler support or each separate analysis can be disabled. In such cases, triggered by the arguments passed to the MLCA Optimizing Compiler, the task functions are again processed by the C-Compiler and sent to the Sarek-Compiler, but only the selected analyses are performed and the selected pragmas are inserted.

In the remainder of this section, we present the alterations made to the IPL and IPA phases of ORC, in order to obtain the necessary inter-procedural data-flow analysis for the structure fields and the inter-procedural array section analysis.

A.4.1 Inter-procedural Structure Fields Analysis

The aim of the Inter-procedural Structure Fields Analysis in the C-Compiler is to compute the defined and used fields of the structures that are input arguments to the task functions. These structures are accessed by multiple tasks and usually represented by a variable of type pointer-to-structure.

Because the original inter-procedural data-flow solver of the IPA analyzes the definitions and uses only for the global variables, during IPL, every input and output argument of task functions, which are of type pointer-to-structure, are promoted to global. After the data-flow analysis is completed, they are transformed back to their original scope, in order to keep the task functions unmodified.

Furthermore, the original inter-procedural data-flow solver of the ORC, does not analyze the definition and uses of the structure fields. In fact, when a field of a structure is accessed via a variable (of type structure or pointer-to-structure), the variable is only marked as indirectly accessed, without the information of the accessed field. In order to solve this problem, several modifications are implemented in IPL and IPA.

First, in IPL, every structure field in the type table is given a unique id for each structure the field may be directly or indirectly accessed from. These ids are used to keep track of the defined and used fields of the structures. A field of a structure may have multiple ids if the pointer of the structure is a field of other structures. Second, the intra-procedural data-flow analyzer of IPL is modified to compute definitions and uses for the structure fields accessed via structure pointers. Finally, each global and formal variable of type pointer-to-structure (including the input and output arguments of tasks) is annotated with the ids of the defined/used structure fields of the corresponding variable. These annotations are inserted to the output object code together with the original local data-flow and control-flow analyses result summaries.

Second, in IPA, the ids of the structure fields are re-computed in such a way that they match with the ones used in IPL. In this way, similar to how definitions and uses are computed for the global variables, for each global and formal variable, the ids of the structure fields definitions and uses are propagated from the bottom of the call graph towards the top of the call graph[2]. Once the ids of the defined and used structure fields are propagated up to the control program and the task functions, they are transformed to Sarek pragma statements in the form of Whirl nodes and inserted to the Whirl tree of the task functions.

[2]The top most node of the call graph is the control program, and the leaves of it are the task functions.

In order for the inter-procedural data-flow solver to produce the exact data-flow information for each task, it should be flow dependent. In other words, a use `u` of a structure field `f` should not be marked as used in a task `t`, if definitions of `f` precede `u` in every control path from the start of `t` to `u`. This ensures the computation of only the upward exposed uses of structure fields. For definitions, such a condition is not necessary, because every definition of a variable or a structure field will be downward exposed, which is the information that the Sarek code transformations need for defined structure fields.

The implemented inter-procedural data-flow solver for the structure fields is a conservative analysis because it is flow independent. In IPL, the intra-procedural data-flow solver is flow-dependent; however, in IPA, the propagation of the definitions and uses of the structure fields on the call graph is flow-independent. In fact, in IPA, a use of a structure field is propagated from a leaf node to a parent node on the call graph, even if the use of the structure field is dominated by a definition of it in the parent node. In other words, if a structure field is marked as used in a function `f1`, it will be marked as used in all the functions `f2, f3, ..., fn` that call `f1`, whether a definition the structure field dominates all the calls to `f1` or not.

This flow-independent nature of the inter-procedural data-flow solver in IPA results in extra `Uses` Sarek pragmas for the structure fields that do not actually have upward exposed uses. The effect of these extra pragmas on the execution cycles of some multimedia applications is investigated in Chapter 8. As the main motivation behind this work is to implement an optimizing compiler for the MLCA, the implementation of a flow dependent data-flow solver for structure fields is left as a future work. As the compiler technology advances such compiler transformations will be available.

A.4.2 Inter-procedural Array Section Analysis

The aim of the inter-procedural array section analysis in the C-Compiler is to generate the defined and/or used sections of the arrays that are input arguments to tasks functions.

Similar to the inter-procedural data-flow solver, the inter-procedural array-section analyzer of the ORC considers only global arrays. Furthermore, for definition/use sections of a global array to be generated, the accesses to this global array should be via `OPR_ARRAY` operations of Whirl. This operation is only generated, by the frontend, for only array accesses in Fortran and for the global arrays accessed using brackets `[` and `]` in C. In fact, the array-section analyzer of IPA is designed for Fortran programs and it is successful in analyzing C programs in case the syntax of the array accesses is similar to that of Fortran. As a result, array accesses using pointer arithmetic and accesses to arrays inside structures are ignored by the IPA. Because substantial modification is required to solve these problems of IPA array-section analysis, we left the implementation of an aggressive array-section analyzer as future work and concentrated our work to obtain array sections for regular accesses (without using pointer arithmetic) of stand-alone (not contained in structures) arrays. For this purpose, intra-procedural array-section analyzer in IPL is altered in order to process such array accesses. On the other hand, the original inter-procedural array section propagation algorithm of IPA is used without any modification.

The MLCA Optimizing Compiler requires the array sections for the arrays that are communicated among tasks[3]. Such arrays have two properties. First, they are accessed via a pointer. For ease of implementation, we assume that such arrays are accessed via a variable of type pointer-to-scalar (eg. `int *buf`) or pointer-to-array (eg. `int (*buf)[]`). Second, because tasks need addresses to access arrays, these pointers are input arguments to the corresponding tasks.

The inter-procedural array-section analysis of ORC is originally designed similar to the other inter-procedural analyses of ORC. In other words, IPL processes each function one by one, i.e. performs intra-procedural array-section analysis, and generates a summary of the array sections. Then, IPA takes these array-section summaries and propagates them

[3]Statically defined local arrays or dynamically created temporary arrays, accessed inside a single task, are ignored.

to the caller functions in the call graph. Since the original array-section propagation algorithm of IPA is satisfactory, we made no modification on the IPA in terms of array-section analysis. However, in order to obtain the required array sections that will be propagated in IPA, the following modifications are implemented in IPL. First, for every task function, input arguments of type pointer-to-scalar or pointer-to-array, referred as *buffer arguments*, are promoted to global, in order to enable array-section analysis for these arguments. Again, after the intra-procedural analysis of array sections is completed in IPL, the scopes of the buffer arguments are transformed back to their original. Second, the Whirl tree of each function is transformed to Fortran style. In other words, we make sure that arrays of concern, i.e. buffer arguments, are accessed via `OPR_ARRAY` operations of Whirl. For this purpose, whenever we see an indirect access (usually with `OPR_ILOAD` operation) to the buffer arguments, we transform the Whirl tree of the access to a Whirl tree that uses `OPR_ARRAY` operation.

We run the original array section analyzer of IPL and obtain the array sections for each function. During IPA, these array sections are propagated towards the task functions in the call graph. Finally, at the end of IPA, we transform each array section definition/use to Sarek `Defs/Uses` Pragma statements and insert these to the Whirl tree of the corresponding task function.

Bibliography

[1] 3SOC documentation - 3SOC 2003 hardware architecture, cradle technologies, Inc., march 2002.

[2] 3SOC programmer's guide, cradle technologies, inc., march 2002. http://www.cradle.com.

[3] Alain Mellan, *Personal communication*, 2003.

[4] The mad. `http://www.underbit.com/products/mad/`.

[5] The MediaBench. `http://cares.icsl.ucla.edu/mediabench/applications.html`.

[6] The OpenMP. `http://www.openmp.org`.

[7] The ORC Compiler. `http://ipf-orc.sourceforge.net`.

[8] The Picochip. `http://www.picochip.com`.

[9] Special issue on system-on-chip. In *IEEE Micro*, Spetember 2002.

[10] V. Balasundaram and K. Kennedy. A technique for summarizing data access and its use in parallelism enhancing transformations. In *PLDI '89: Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, pages 41–53. ACM Press, 1989.

[11] A. Deutsch. Interprocedural may-alias analysis for pointers: beyond k-limiting. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 230–241, New York, NY, USA, 1994. ACM Press.

[12] R. Eigenmann, J. Hoeflinger, Z. Li, and D. Padua. Experience in the automatic parallelization of four perfect-benchmark programs. In *Proceeding of the 4*th *Workshop on Programming Languages and Compilers for Parallel Computing*, August 1991.

[13] C. Nicol et. al. A single-chip, 1.6-billion, 16-b MAC/s multiprocessor DSP. *IEEE Journal of solid-state circuits*, 35(2), March 2000.

[14] The Parallel Computing Forum. PCF parallel fortran extensions. *SIGPLAN Fortran Forum*, 10(3):1–57, 1991.

[15] J. Fritts, W. Wolf, and B. Liu. Understanding multimedia application characteristics for designing programmable media processors. In *Proceedings of Media Processors*, pages 2–13, January 1999.

[16] J. Gu, Z. Li, and G. Lee. Symbolic array dataflow analysis for array privatization and program parallelization. In *Proceedings of ACM/IEEE Conference on Supercomputing*, page 47, December.

[17] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach, 3rd ed.* Morgan Kaufmann, 2003.

[18] M. Hind. Pointer analysis: haven't we solved this problem yet? In *PASTE '01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 54–61, New York, NY, USA, 2001. ACM Press.

[19] M. Hind, M. Burke, P. Carini, and J. Choi. Interprocedural pointer alias analysis. *ACM Trans. Program. Lang. Syst.*, 21(4):848–894, 1999.

[20] F. Karim, A. Mellan, A. Nguyen, U. Aydonat, and T.S. Abdelrahman. The hyperprocessor: A template system-on-chip architecture for embedded multimedia applications. *In Proceedings of the Workshop on Application Specific Processors*, pages 66–73, 2003.

[21] F. Karim, A. Mellan, A. Nguyen, U. Aydonat, and T.S. Abdelrahman. A multilevel computing architecture for multimedia applications. *IEEE Micro*, 24(3):55–66, 2004.

[22] C. Lee, M. Potkonjak, and H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *Proceedings of the International Symposium on Microarchitecture*, pages 330–335, 1997.

[23] S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1997.

[24] Y. Paek, J. Hoeflinger, and D. Padua. Simplification of array access patterns for compiler optimizations. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming language design and implementation*, pages 60–71. ACM Press, 1998.

[25] R.Rajsuman. *System-on-a-chip: Design and Test.* Artech House, 2000.

[26] L. J. Toomey, E. C. Plachy, R. G. Scarborough, R. J. Sahulka, and J. F. Shaw. IBM parallel Fortran. *IBM Syst. J.*, 27(4):416–435, 1988.

[27] P. Tu. *Automatic Array Privatization and Demand Driven Symbolic Analysis.* PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1995.

[28] D. Wall. Limits of instruction-level parallelism. In *Proc. of the Int'l Conf. on Architectural Support for Programming Languages and Operating System (ASPLOS)*, pages 176–189, 1991.