Automatic Task Formation Techniques for the Multi-Level Computing Architecture

by

Kirk Stewart

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

Copyright © 2008 by Kirk Stewart
Abstract

Automatic Task Formation Techniques for the Multi-Level Computing Architecture

Kirk Stewart
Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto
2008

The Multi-Level Computing Architecture (MLCA) is a multiprocessor system-on-chip architecture designed for multimedia applications. It provides a programming model that simplifies the process of writing parallel applications by eliminating the need for explicit synchronization. However, developers must still invest effort to design applications that fully exploit the MLCA’s multiprocessing capabilities. We present a set of compiler techniques to streamline the process of developing applications for the MLCA. We present an algorithm to automatically partition a sequential application into tasks that can be executed in parallel. We also present code generation algorithms to translate annotated, sequential C code to the MLCA’s programming model. We provide an experimental evaluation of these techniques, performed with a prototype compiler based upon the open-source ORC compiler and integrated with the MLCA Optimizing Compiler. This evaluation shows that the performance of automatically generated code compares favourably to that of manually written code.
Acknowledgements

I would like to express my deep gratitude to all the people who have made this work possible. Professor Tarek S. Abdelrahman provided constant guidance and feedback throughout the course of my research, and his high standards are hopefully reflected in my thesis. I would also like to thank Utku Aydonat for his excellent work on the MLCA Optimizing Compiler. It is a crucial link in the tool-chain described here, and Utku’s constant willingness to provide information, enhancements and bug fixes (even while on vacation!) is greatly appreciated.

I would like to thank Alain Mellan and Faraydon Karim for their roles in the design and development of the MLCA itself. In particular Alain wrote and maintained the MLCA simulator, which proved to be an invaluable tool during the development and evaluation of my work.

Finally I would like to thank my family and friends for their unwavering support and encouragement. My parents, Glenn and Helga, provided many words of wisdom, as did my in-laws Jean and James McGuire. My wife Kerry’s devotion and support throughout this work has been an inspiration – without her it would not have been possible.

This work was supported by grants from the National Sciences and Engineering Research Council of Canada (NSERC), the Ontario Centre of Excellence for Communications and Information Technology (CITO), and STMicroelectronics.
## Contents

1 **Introduction** 1

1.1 Contribution 3

1.2 Thesis Organization 4

2 **Background** 6

2.1 Control and Data Flow 6

2.2 Superscalar Processors 9

2.2.1 Data Dependence and Register Renaming 10

2.3 Function Outlining 11

2.4 Points-To Analysis 12

2.4.1 A Points-To Analysis Algorithm 14

3 **The Multi-Level Computing Architecture (MLCA)** 17

3.1 Architecture 17

3.2 Programming Model 19

3.2.1 The Control Program 20

3.2.2 Task Bodies 20

3.2.3 Parallel Execution 22

3.3 Compiler Support 24

3.3.1 Shared Aggregate Data 25

3.3.2 Task Parameter Deaggregation 26
List of Tables

7.1 Parameters used for the HFM in all experiments. . . . . . . . . . . . . . 89
7.2 Effects of Useless Output Removal . . . . . . . . . . . . . . . . . . . . 106
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Control-Flow Graph</td>
<td>7</td>
</tr>
<tr>
<td>2.2</td>
<td>Data-Flow Graph</td>
<td>8</td>
</tr>
<tr>
<td>2.3</td>
<td>A Simple Superscalar Architecture</td>
<td>9</td>
</tr>
<tr>
<td>2.4</td>
<td>True and False Data Dependences</td>
<td>10</td>
</tr>
<tr>
<td>2.5</td>
<td>Function Outlining</td>
<td>12</td>
</tr>
<tr>
<td>2.6</td>
<td>Context- and Flow-Sensitivity in Alias Analysis</td>
<td>14</td>
</tr>
<tr>
<td>2.7</td>
<td>Alias Analysis Algorithm Example</td>
<td>15</td>
</tr>
<tr>
<td>2.8</td>
<td>Conservative Alias Analysis</td>
<td>16</td>
</tr>
<tr>
<td>3.1</td>
<td>Architectural Block Diagrams</td>
<td>18</td>
</tr>
<tr>
<td>3.2</td>
<td>A Simple Sarek Program</td>
<td>21</td>
</tr>
<tr>
<td>3.3</td>
<td>Execution Diagram</td>
<td>23</td>
</tr>
<tr>
<td>3.4</td>
<td>MOC Flow Diagram</td>
<td>24</td>
</tr>
<tr>
<td>4.1</td>
<td>Porting a basic C program</td>
<td>31</td>
</tr>
<tr>
<td>4.2</td>
<td>Compiler Tool-Chain</td>
<td>32</td>
</tr>
<tr>
<td>5.1</td>
<td>Basic split pragma usage</td>
<td>36</td>
</tr>
<tr>
<td>5.2</td>
<td>Split pragma in a loop body</td>
<td>37</td>
</tr>
<tr>
<td>5.3</td>
<td>Task generation algorithm</td>
<td>39</td>
</tr>
<tr>
<td>5.4</td>
<td>An example of automatic task generation</td>
<td>43</td>
</tr>
<tr>
<td>6.1</td>
<td>Task splitting algorithm</td>
<td>45</td>
</tr>
</tbody>
</table>
7.8 GSM Results Without Using Scope to Refine Points-To Analysis . . . . . 108
Chapter 1

Introduction

The proliferation of portable electronic devices such as cellular phones and personal digital assistants (PDAs) has driven rapid advances in the area of embedded system design. Such systems are characterized by requirements for high performance with very low power consumption, as well as tight restrictions on physical size and a need for rapid time to market [23]. Systems on a chip (SoCs), which incorporate multiple components such as general purpose processors, digital signal processors (DSPs) and memory on a single chip are therefore an ideal paradigm for such applications. They provide a wide range of functionality in a single package and, by using modular component designs, can allow for the rapid development of a special-purpose chip. Furthermore, they reduce the need for off-chip communication which is one of dominant contributors to overall system power consumption [22].

The Multi-Level Computing Architecture (MLCA) is a novel SoC architecture [20]. It provides high performance and low power consumption through the inclusion of multiple programmable processing units and a shared on-chip memory. Its primary novelty is the simplicity of its programming model – it allows the exploitation the MLCA’s multi-processing capabilities without the need for explicity inserted synchronization and data communication. The programmer must only divide an application into coarse-grained
Chapter 1. Introduction

execution units, or tasks, and express the control and data flow among tasks. The MLCA includes a control processor to coordinate the execution of tasks at run-time. When possible, the control processor dispatches tasks in parallel and even out of order, using the same techniques that a superscalar processor employs to exploit fine-grained instruction-level parallelism.

An MLCA program consists of two major components: a control program and a set of task bodies. The former is written in a special-purpose C-like language called Sarek \[20\], and describes the inter-task control and data flow. The latter can be written in any language that can be compiled to execute on the MLCA’s processing units; this is generally a high-level language such as C. Development of a new MLCA application is relatively straightforward as the control program and task bodies can be built simultaneously, allowing the programmer to incrementally express the application’s control and data flow throughout development. Porting an existing sequential application can be more complicated. The code must be refactored into tasks, and a Sarek program must be developed to accurately express the program’s control and data flow. This process can be accomplished manually, but it can be greatly simplified and streamlined by proper compiler support.

This thesis provides a set of techniques for the compiler-driven translation of sequential applications to the MLCA programming model. It presents an iterative, heuristic-based algorithm to partition a program into tasks such that a significant degree of parallelism is exposed to the MLCA hardware. The algorithm requires a minimal degree of programmer involvement but is also designed to be transparent, allowing a programmer to tune and provide feedback throughout its execution. Partitions created by this algorithm are represented in the original program source by a set of directives, and can therefore be easily modified. The task formation algorithm differs from existing automatic parallelization techniques in its top-down iterative approach, and its heuristic use of common characteristics of multimedia applications. These aspects make it uniquely
suited to the novel execution environment that the MLCA provides.

This thesis also presents a code generation algorithm, whereby annotated sequential source code is translated to a valid MLCA program. This process involves refactoring the original source into a set of task bodies and generating a corresponding Sarek control program. To ensure the correctness of the resulting application the Sarek program must accurately represent the original program’s control and data flow. Therefore, the code generation algorithm incorporates a variety of analyses to support this requirement.

The result of this work is a compiler framework that allows a programmer to rapidly explore and prototype a large number of program partitions, thereby easing the task of porting and tuning an existing application for execution on the MLCA. The task formation algorithm and the code generation algorithm can be used together or in isolation, giving the programmer full control over the development process.

Additionally, this thesis presents an implementation and evaluation of these techniques in a prototype compiler. This prototype has been integrated with the MLCA Optimizing Compiler [12] to provide a full compiler tool-chain targeting the MLCA architecture. We evaluate the performance of this tool-chain on six multimedia applications. In most cases the automatic task formation and code generation algorithms succeed in producing competitive results when compared to manually ported and optimized versions of the applications.

1.1 Contribution

The major contributions of this thesis are summarized as follows:

- An Automatic Task Formation Algorithm – We describe an iterative, heuristic algorithm to automatically partition a sequential program into tasks. The algorithm works to partition code into separate tasks and to control task granularity such that the resulting program exhibits good performance when executed on the MLCA.
• **Code Generation Algorithms** – We describe a set of algorithms to refactor an existing program into task bodies and a Sarek control program. These algorithms perform control and data flow analyses to ensure that the correctness of the resulting MLCA program is preserved while minimizing the introduction of false data dependencies that can limit program performance.

• **A Program Annotation System** – We introduce a directive to represent task boundaries in otherwise unmodified sequential source code. We define its semantics during the translation process between original source code and the MLCA programming model.

• **Implementation and Evaluation** – We have implemented the techniques described in this thesis and have integrated them into the existing MLCA tool-chain. We have shown through experimental evaluation with real multimedia programs that these techniques provide performance that is competitive with manually generated code.

### 1.2 Thesis Organization

Chapter 2 presents a brief overview of the central concepts that underly this work, including program control and data flow, superscalar processors, function outlining and alias analysis. Chapter 3 is a description of the MLCA architecture and programming model, as well as the optimization techniques employed by the MLCA Optimizing Compiler. An overview of the Automatic Task Formation framework that we have developed for the MCLA is presented in Chapter 4. Chapter 5 presents our algorithm for automatically partitioning an existing program into tasks. Chapter 6 contains a detailed description of code generation techniques that can be used to translate sequential programs to the MLCA programming model. Chapter 7 describes our experimental framework, and includes the results of our evaluation. Chapter 8 summarizes related work. Finally, Chapter
provides a conclusion and some directions for future work.
Chapter 2

Background

This chapter presents an overview of several basic concepts that underly this work. These include control and data flow, superscalar processor architectures, function outlining and points-to analysis.

2.1 Control and Data Flow

A program is a collection of statements, and the set of rules that define the order in which those statements are executed is referred to as the control flow of the program. Statements are generally stored in a linear list, and when the execution of one statement is complete the next statement in the list may be executed. Most languages provide special control-flow statements that allow the programmer to augment this linear flow. The complexity of these statements varies among languages. Lower level assembly-type languages usually only provide simple branch statements. Higher level languages, such as C, provide more complex control-flow statements such as loops, switches and conditionals.

It is sometimes useful to represent a program as a control-flow graph, or CFG. The program’s statements are grouped together into basic blocks such that statements in a basic block are executed in a strictly linear order. Furthermore, it must be possible for execution of a basic block to begin only with the first statement and to end only with the
last statement. Figure 2.1(a) contains a fragment of assembly-like code that calculates $2^8$. This code has been divided into three basic blocks: the first boundary, between lines 2 and 3, is made necessary by the fact that line 3 is a branch target. Since a basic block must only be entered at its first statement all branch targets must be the start of a basic block. Similarly, since the only exit point of a basic block may be its last statement, the branch on line 6 makes that line the end of the block.

Basic blocks are the vertices of a control-flow graph. Edges in the graph represent control flow between blocks. Figure 2.1(b) shows the CFG for the previous code fragment. Execution of line 2 is always followed by execution of line 3, so there is a single edge from BB1 to BB2. Line 6 is a conditional branch statement, so its execution will be followed by that of line 3 or line 7 depending on the value in r2. There are therefore two outgoing edges from BB2: one to itself and one to BB3.

Another characteristic of a program is the manner in which information is passed between statements. While a program’s control flow is expressed explicitly in most languages through control-flow statements, its data flow is often expressed implicitly. A
statement may write data to a specific location, and a subsequent statement may read data from that location. This data location may be a register, a named memory location (eg. a stack variable), or an unnamed memory location (eg. memory allocated on the heap). Like control flow, data flow can be represented as a graph. Depending on the desired resolution vertices may be either statements or basic blocks. Each edge in the graph represents the flow of a single value from one vertex to another.

Figure 2.2 shows a version of the control-flow graph from figure 2.1 that has been modified to include data-flow edges. Two values (stored in \(r1\) and \(r2\)) are produced in BB1 and consumed in BB2, so there are two edges from BB1 to BB2. BB2 also produces two values, which are consumed by future instances of BB2. There are thus two edges from BB2 to itself.
2.2 Superscalar Processors

Most processor architectures expect programs to be expressed as a stream of instructions that are intended to be processed sequentially. Often, however, a purely sequential execution schedule is not necessary – if two instructions operate on mutually exclusive data they do not have to execute in the original program order. In fact, they may even be executed in parallel if the hardware is capable of doing so. Modern processor architectures go to great lengths to identify and exploit this instruction-level parallelism (ILP). Architectures that do so are said to be superscalar [18].

Figure 2.3 shows a simplified block diagram for a superscalar processor. A Fetch/Decode unit reads instructions from memory in program order. It examines the data dependencies between instructions and dispatches them to the execution units as soon as all of their input dependencies are satisfied. The processor has multiple execution units that may work in parallel. Thus if two instructions are independent of each other, and there are free execution units to process them, they can be dispatched at the same time.
2.2.1 Data Dependence and Register Renaming

Care must be taken when performing out-of-order execution that a program’s data dependencies be observed. If two instructions operate on the same data location (ie. register) then they must be executed in the original program order to guarantee correctness. Data dependencies therefore limit the amount of instruction-level parallelism present in a program. Fortunately, it is possible to divide data dependencies into two categories: true and false dependencies. The former are caused by a true flow of data between instructions and are therefore unavoidable. The latter result from the re-use of registers to store multiple, independent values at different program points. They are not necessitated by the flow of data through the program – if more registers were available in the register file then such dependencies could be avoided.

Figure 2.4 demonstrates both true and false data dependencies. Instruction number 3 uses values produced by instructions 1 and 2, so it may only execute once those instructions have completed. Those dependencies result from a real flow of data, therefore they are true dependencies. Like instruction 3, instruction 4 writes a value to r3. It must therefore wait until instruction 3 has finished before it may execute; if it did not then it is possible that instruction 3 could perform its write after instruction 4 and a stale value would be stored in r3. This dependence is not the result of an actual flow of data – instruction 4 does not use any data produced by instruction 3. Rather, it is caused by the reuse of r3. If instruction 4 wrote to a different register, r4, then it would no longer have to wait for instruction 3 to preserve the program semantics.

More formally, dependencies between two instructions can be divided into four differ-
Chapter 2. Background

ent classes:

- **Read After Write (RAW)** – An instruction reads a register after a prior instruction writes to that register.

- **Write After Read (WAR)** – An instruction writes to a register after a prior instruction reads from the register.

- **Write After Write (WAW)** – An instruction writes to a register after a prior instruction writes to that register.

- **Read After Read (RAR)** – An instruction reads a register after a prior instruction reads that register.

RAW dependencies are true data dependencies – they result from an instruction using data that is produced by a prior instruction. WAR and WAW dependencies are false data dependencies. They are caused only by the fact that instructions reuse registers that are used by prior, otherwise independent instructions. RAR dependencies are included above for completeness, but generally do not require instructions to be serialized since neither instruction modifies the data in the shared register.

A superscalar processor can increase the instruction-level parallelism available in a program by eliminating false dependencies. This can be done through a process called **register renaming**. When a false dependence is detected between two instructions the later instruction is changed to write to a different register. By renaming its output register the false dependence is broken, and the two instructions can safely be executed out of order. All future instructions that use the result calculated by this instruction must also be modified to read the renamed register.

2.3 Function Outlining

*Function outlining* is a simple code transformation that extracts a section of a function
Chapter 2. Background

void myfunc(int a, int *b) {
    int i, j, sum;
    sum = 0;
    for (i = 0; i < 1000; i++) {
        sum += b[i];
        if (a) {
            for (j = i; j >= 0; j--)
                printf("%d\n", b[j]);
        }
    }
    printf("sum is %d\n", sum);
    return;
}

(a) Before Outlining

Figure 2.5: Function Outlining

body and encapsulates it in a newly created function. The original code is then replaced with a call to the newly created function. The name is derived from the fact that this transformation is the logical opposite of function inlining, in which function calls are replaced by the body of the callee. Figure 2.5 presents an example of the outlining transformation.

Outlining introduces a small overhead as it increases the number of function calls in a program. If used judiciously, however, it can provide a performance enhancement. Moving infrequently used code paths to a separate function can reduce the size of frequently used sequences. This size reduction may make it feasible to inline the hot code path, and may also improve instruction cache locality by allowing the hot path to fit in a smaller number of cache lines [33].

2.4 Points-To Analysis

C, like many other languages, allows indirect data access through the use of pointers. It is therefore possible for a single statement to access a different data location each
time it is executed. Pointers are a powerful programming tool, but their use can pose a significant problem for compilers. It is often necessary for a compiler to know what data locations are accessed by each statement in a program. To do so the compiler must reason about what data locations a pointer could point to when it is dereferenced. This process is called *points-to analysis* [19].

A fundamental tradeoff when designing a points-to analysis algorithm is that of time complexity versus precision. The worst-case time complexities of existing algorithms range from linear to exponential [19]. The faster algorithms typically provide fairly imprecise, conservative results – they indicate that a pointer may target some locations that could be excluded with more in-depth analysis.

Two major attributes of points-to analysis algorithms that affect this tradeoff between complexity and precision are *flow*- and *context-sensitivity*. Algorithms that are flow-sensitive incorporate intraprocedural control-flow information into their analyses. Algorithms that are context-sensitive incorporate information about the calling context into their analyses when examining each function. Flow- and context-sensitivity can lead to significantly more precise results, but they also require more complex (and therefore slower) algorithms.

Figure 2.6 demonstrates the difference that flow- and context-sensitivity can make on the precision of an algorithm. By inspection it can be seen that the `printf` statement in `main` will always print the value of `a`, since it is protected by the same condition that causes the address of `a` to be stored in `ptr`. A flow-sensitive algorithm would recognize this fact, but a flow-insensitive algorithm would not. Since it would not incorporate control-flow information into its analysis it would conservatively assume that `ptr` could potentially point to both `a` and `b` when it is dereferenced in the `printf` statement.

The `printf` statement in the function `foo` also dereferences a pointer. At different times that pointer targets both `a` and `b`, but there is no single program point at which it could potentially target both. A context-sensitive algorithm would compute two separate
int main() {
    int a, b;
    int *ptr;

    if (cond)
        ptr = &a;
    else
        ptr = &b;

    if (cond)
        printf("%d", *ptr);
    foo(&a);
    foo(&b);
}

void foo(int *arg) {
    printf(" %d ", *arg);
}

Figure 2.6: Context- and Flow-Sensitivity in Alias Analysis

solutions for foo, one for each callsite. The first solution would show that arg points only to a, and the second would show that arg points only to b. A context-insensitive algorithm would compute a single whole-program solution. It would therefore indicate that arg points to both a and b.

2.4.1 A Points-To Analysis Algorithm

One example of a points-to analysis algorithm is Steensgaard’s Points-to Analysis in Almost Linear Time algorithm [30]. The algorithm works by placing each statement that accesses memory (either directly or indirectly) in an alias class. Each alias class initially contains exactly one statement. The algorithm then iterates across the program, merging alias classes whenever it finds a statement that creates an alias between two classes.

A simple example of the algorithm’s execution is shown in Figure 2.7. All statements that access memory are initially assigned distinct alias classes, then the algorithm merges alias classes as necessary. Alias classes 4 and 5 are merged because they both include ptr. Alias classes 6, 7 and 8 are merged, however this is done in two steps. First classes 6 and 7 are merged, because a and *ptr may, under some circumstances, refer to the
int main() {
    int x, y;
    myfunc(x, &y);
    printf("%d", x);        // AC 1
    printf("%d", y);        // AC 2
}

void myfunc(int myarg, int *myptr) {
    int a, b, c;
    int *ptr;
    if (my_arg) {
        ptr = &a;            // AC 3
    } else {
        ptr = &b;            // AC 4
    }
    printf("%d", a);        // AC 6
    printf("%d", *ptr);     // AC 7
    printf("%d", b);        // AC 8
    printf("%d", c);        // AC 9
    printf("%d", *myptr);   // AC 10
}

(a) Initial Alias Class Assignment  
(b) After Algorithm Completion

Figure 2.7: Alias Analysis Algorithm Example
same memory location. Then classes 7 and 8 are merged because *ptr and b may refer to the same memory location. Since classes 6 and 7 have already been merged, this second merge operation results in all three statements being placed in alias class 6. It is interesting to note that classes 2 and 10 are merged, but classes 1 and 3 are not. This is because x is passed to myfunc by value, while y is passed by reference.

With some consideration it can be seen that this algorithm is flow- and context-insensitive. It provides a whole-program solution: if a single pointer is used to point at two separate variables, then all accesses to the variables (as well as all indirect accesses through that pointer) will be in the same alias class. Even if the pointer can never simultaneously point to both variables (as a result of program control- or data-flow) this will be true. Figure 2.8 provides an example of a situation in which this conservatism causes imprecise results. ptr points at both x and y at separate times in the program, but it can never point at both simultaneously. It is therefore technically possible to determine at each dereference exactly which variable ptr refers to, and no merging of alias classes should be necessary. The algorithm, as designed, does so however.

The output of the algorithm is a mapping from program statements to alias classes. If two statements are in the same alias class, then there is a possibility (but not a guarantee) that they access the same memory location.
Chapter 3

The Multi-Level Computing Architecture (MLCA)

The Multi-Level Computing Architecture, commonly known as the MLCA, is a multi-processor system-on-chip computing architecture. It is targeted at embedded multimedia applications, which are often characterized by high levels of coarse-grained parallelism, and by low power requirements. As such, the MLCA is a template architecture: it is a minimal specification of an interprocessor work-sharing and communication model, with an accompanying instruction set. This allows instances of the MLCA to be specialized for their specific application. The number and type of processing units can be varied to suit application requirements, as can the size and layout of memory. This customizability allows system designers to maximize efficiency by including only the hardware that their application needs.

3.1 Architecture

The basic unit of computation on the MLCA is a task. Tasks are a coarse-grained unit – they can perform complex operations, and generally have execution times ranging from several hundred cycles to tens of thousands of cycles. Tasks are analogous to (and are
Chapter 3. The Multi-Level Computing Architecture (MLCA)

A minimal MLCA configuration includes one or more processing units (PUs), a universal register file (URF), and a single control processor. PUs execute all task bodies, and can be general-purpose processors (such as PowerPC or ARM cores) or they can be special-purpose units (such as FPGAs or DSPs). A single MLCA system may have a heterogeneous set of PUs. The URF contains a set of registers, which can be read and written by tasks. The MLCA architecture does not specify any particular structure for the URF – registers may be fixed- or variable-length – however partial register access is not possible. A task must read or write a URF register in its entirety. The control processor executes a control program, which consists of a sequence of control-flow and task instructions. Each task instruction specifies a task to be executed as well as an explicit listing of the URF registers that it will read and write.

Figure 3.1(a) shows a block diagram of the MLCA architecture, and Figure 3.1(b) shows a diagram for a traditional superscalar processor. It can be seen that the two are virtually identical. The principal difference is the granularity of computation – the MLCA executes tasks in the same way that a superscalar processor executes instructions.

A task instruction is considered ready to execute when all prior writes to its input
registers have completed. When a task is ready to execute it is passed to the control processor’s task scheduling unit, which dispatches it to a PU. As in a superscalar processor, the control processor can perform URF register renaming to break false data dependencies, and it is capable of out-of-order task execution. These aspects of the MLCA are discussed in further detail in Section 3.2.3.

Aside from the URF, the memory architecture of the MLCA is intentionally undefined. It could incorporate a global memory that is shared between all PUs, local memory for each PU, or a combination of both global and local memories. Likewise, the interconnect between the PUs, URF and memory could be a shared bus, a hierarchical bus, or a crossbar. The MLCA is a template architecture – it is intended for use in embedded applications, which generally have tight constraints upon cost, performance and power consumption. Leaving details such as the number and type of PUs, and the memory and bus architectures undefined allows the MLCA to be customized according to the specific requirements of each application to which it is applied.

For the purposes of this work we use an MLCA instance with 32-bit fixed-length URF registers, and a large global shared memory.

3.2 Programming Model

An MLCA program consists of two parts: a control program and a set of task bodies. The control program is implemented in the native assembly language of the control processor, Hyper Assembly (HASM). The task bodies are implemented in the native machine language of the PUs on which they are intended to run. It is possible for a programmer to write an MLCA program entirely in assembly, however in practice high-level programming languages are generally used. The control program can be written in a specially developed language called Sarek, and task bodies are often implemented in standard C.
3.2.1 The Control Program

Sarek is a high-level, C-like programming language developed specifically for the MLCA. Most Sarek statements are simple task calls, however Sarek does include several high-level control flow constructs such as conditional and loop statements. Two data types are available to the programmer: data and control registers. The former is used for all data flow between tasks, and the latter must be used for the calculation of control flow conditions. Task calls are syntactically similar to C function calls. They specify the name of the task to be called, a list of data registers that the task will read, a list of data registers that the task will write and, optionally, a control register that the task will write.

Figure 3.2(a) shows a simple Sarek program that computes the first 32 powers of 2. It demonstrates the use of control registers, data registers, task calls and looping constructs.

It should be noted that while Sarek bears many similarities to C it provides a subset of C’s features. It does not include, for example, facilities for function calls and for the evaluation of arbitrary expressions. This is because Sarek programs are executed on the MLCA’s control processor. The control processor is designed for a specific purpose – to perform task scheduling and URF register renaming – and as such does not include many of the features found in general-purpose processors, such as an ALU and a memory-management unit.

3.2.2 Task Bodies

Task bodies can be implemented in any language that can be compiled for execution on the target PUs. In practice, they are generally written in C. Figure 3.2(b) shows the task bodies that correspond to the example Sarek program. Each task is implemented by a C function. Task parameters are specified as input and output URF registers, and these semantics do not map well to the function parameter passing conventions in
main() {
  cr_t more_work;
  reg_t pow, cur;

  more_work = Init(out cur);

  while (more_work) {
    more_work = Increment(in cur, out pow, out cur);
    Calc_Pow(in pow);
  }
}

(a) Sarek Control Program

#define NUM_POWS 32

int Init() {
  writeArg(0, 0); // initialize cur
  return 1;       // initialize more_work
}

int Calc_Pow() {
  int i;
  int val = 1;
  int pow = readArg(0);

  for (i = 0; i < pow; i++)
    val *= 2;
  printf("2^%d = %d\n", pow, val);
  return 0;
}

int Increment() {
  int cur = readArg(0);
  writeArg(0, cur);
  writeArg(1, ++cur);
  if (cur < NUM_POWS)
    return 1;
  else
    return 0;
}

(b) Task Bodies

Figure 3.2: A Simple Sarek Program
C. As a result, functions that implement task bodies do not have any formal parameters. Rather, they access the task arguments through a special API that exports two functions: \texttt{int readArg(int arg\_num)} and \texttt{void writeArg(int arg\_num, int val)}. These functions transfer data between the URF and a PU’s addressable memory. \texttt{readArg} accepts a single parameter, which specifies a position in the task’s input argument list. It reads the URF register specified in that position and returns the value stored in that register. \texttt{writeArg} accepts two parameters: a position in the task’s output argument list and a value. It writes the value to the URF register in the specified position.

A task body must execute exactly one \texttt{readArg} call for each input parameter, and exactly one \texttt{writeArg} call for each output parameter. In addition to performing data transfer, these calls provide synchronization cues to the control processor. By executing a \texttt{readArg} call a task signals that it has locally cached the value stored in a URF register, so any future writes to that register will not violate an anti-dependence with the current task. Likewise, a \texttt{writeArg} call signals to the control processor that it may dispatch any tasks that are waiting for the value of the register being written.

In addition to reading and writing data registers, a task function may optionally write a single control register. This is analogous to the process of a function returning a value in C, therefore it is mapped to the C \texttt{return} statement. This can be seen in the \texttt{Increment} task in the example.

### 3.2.3 Parallel Execution

The explicit specification of task inputs and outputs in each task call allows the control processor to perform register renaming and out-of-order execution. This is demonstrated in Figure 3.3, which shows the execution of the program in Figure 3.2 on an MLCA system with 3 PUs. The \texttt{Increment} task both reads and writes the \texttt{cur} register. This creates a true data dependence between each instance of the task. The control processor recognizes this fact, and does not dispatch an instance of \texttt{Increment} until the previous
instance has completed. Likewise, the data flow through the pow data register creates a true data dependence between the Increment and Calc_Pow tasks in each iteration of the loop. Again, the control processor recognizes this dependence and does not dispatch an instance of Calc_Pow until its preceding instance of Increment has written to pow.

Each instance of Calc_Pow only depends on one corresponding instance of Increment, and it does not depend on any other instances of Calc_Pow. As a result, the control processor is free to dispatch an instance of Calc_Pow before the instances from previous iterations have completed. Furthermore, the control processor can perform register renaming on the cur register, allowing a new instance of Increment to be dispatched before the previous instance’s output value has been consumed by its corresponding instance of Calc_Pow.

In the same manner that a superscalar processor automatically exploits instruction-level parallelism, the MLCA exploits coarse-grained task-level parallelism. As shown in Figure 3.3 this can result in significant amounts of parallel execution, which leads to large
Program speedups.

3.3 Compiler Support

Compilation of an MLCA program can be divided into two main phases: compilation of the control program, and compilation of the task bodies. Each phase offers numerous opportunities for optimization. The MLCA Optimizing Compiler (MOC) is a tool chain developed at the University of Toronto to perform this optimization and compilation.

Figure 3.4 is a flow diagram of the MOC. Its input is the C and Sarek source for an MLCA program. It performs a variety of optimizations on this source (described below) and produces modified C and Sarek as output. A non-optimizing Sarek compiler is then used to produce HASM code suitable for execution on the MLCA, and a separate C compiler is used to produce object code that will run on the PUs. The C compiler is free to perform standard C optimizations during this compilation, which can result in improved application performance by reducing task run-times.
3.3.1 Shared Aggregate Data

A common characteristic of MLCA programs is the sharing of aggregate data structures among tasks. These aggregates may be buffers, or they may be C structs. The use of aggregates introduces two obstacles to efficient parallel execution. The first is false sharing. If two tasks read and write disjoint portions of a single aggregate then they can safely execute in parallel. If the aggregate is represented in the URF as a single register then those two tasks will be forced to serialize as the single register will have to be declared as both an input and an output for both tasks. The false sharing problem is not unique to the MLCA; it is common to virtually all parallel programming environments.

The second obstacle posed by shared aggregates is the fact that, in general, they cannot be assumed to fit entirely in a single URF register. The solution to this problem is to allocate them in shared memory, and to pass a pointer to this memory between tasks that will access them. Unfortunately, doing so effectively disables the superscalar techniques employed by the MLCA to extract parallel performance from the program. Since the aggregate itself is not stored in the URF the MLCA is not able to perform register renaming to break false dependencies between tasks. Furthermore, care must be exercised by the programmer to make sure that the MLCA does not execute tasks that access a common section of an aggregate in parallel. This may be accomplished only by making the pointer to the aggregate both an input and an output to any task that accesses it. Even tasks that perform read-only access must output the pointer, to ensure that any subsequent tasks that write to the aggregate will be dispatched only after the read is complete.

Most of the optimizations performed by the MOC address these problems. Where possible they can enable effective parallel execution of tasks in the face of shared aggregates.
3.3.2 Task Parameter Deaggregation

One optimization performed by the MOC, \textit{task parameter deaggregation}, addresses shared structs. Each individual field of a struct is allocated to the URF, rather than the entire struct as a single unit. Tasks then include only the URF registers for the fields that they use in their access lists. This simultaneously solves both the false sharing problem and the pointer synchronization problem: tasks that access disjoint sections of a shared aggregate will reflect this fact in their parameter lists, and the MLCA hardware will automatically allow them to execute in parallel. Furthermore, since scalar fields of the struct will be allocated to the URF rather than to shared memory, the MLCA can effectively perform register renaming to break false dependencies. De-aggregation can be performed recursively to handle structs within structs. It also exposes buffers within structs to the following buffer optimizations.

3.3.3 Buffer Privatization

As discussed earlier, pointers to shared buffers must be declared as both task inputs and outputs, regardless of the task’s access pattern. Even if the task performs read-only access it must both read and write the pointer to that buffer. This correctly prevents violations caused by WAR, WAW and RAW hazards; however, it forces full serialization of all tasks that access the buffer. In some situations it is possible to perform buffer privatization to allow for correct parallel execution in the presence of a shared buffer.

It is sometimes possible to divide the set of tasks that access a shared buffer into disjoint subsets, such that all tasks in a subset that read data from the buffer have a prior task in the subset that writes data to the same section of the buffer. In other words, all data in a subset is defined before it is used. If such subsets exist then a separate, private copy of the buffer can be allocated for each subset. This allows for the safe parallel execution of tasks in separate subsets, as they will be operating on different
3.3.4 Buffer Replication

A slightly different optimization, buffer replication, can be used to allow parallel task execution in the presence of WAR hazards. If a task writes to a section of a buffer, and a subsequent task performs read-only access, it is possible to create a copy of the buffer for the second task. This allows it to execute in parallel with subsequent tasks that write to the buffer.

3.3.5 Buffer Renaming

Another similar optimization is buffer renaming. Rather than creating a copy of the buffer it works by simply manipulating the URF dependencies of tasks to allow the parallel execution of several read-only tasks. Recall that by default synchronization of buffers allocated in shared memory is accomplished by storing a pointer to the buffer in a URF register. That register is declared as both an input and output for all tasks that access the buffer.

Buffer renaming introduces a second URF register for a buffer. This register is written (but not read) by all tasks that perform read-only access, and is read by all tasks that perform write access. The pointer to the buffer is then declared only as an input to each read-only task. This allows such tasks to execute in parallel, while guaranteeing that writes to the buffer will not occur until all prior reads have completed.

3.4 Benefits of the MLCA

The MLCA presents a number of benefits when compared to other multiprocessor architectures. These include:
• *A Simple Programming Model* – Traditional parallel programming models are quite complex. They require the programmer to explicitly code all data synchronization. This requires a programmer to divide all data locations that may be shared between multiple threads into critical sections, then develop and implement a locking scheme to protect each critical section. In contrast, the MLCA provides a programming model that greatly resembles sequential programming. The programmer must only specify which data locations a task will access, and the hardware will automatically handle all necessary synchronization at runtime.

• *Automatic Exploitation of Parallelism* – The MLCA can effectively and automatically exploit both coarse- and fine-grained parallelism in an application. The former is done by the control processor when it performs URF register renaming and out-of-order task execution. The latter may be performed by the PUs themselves if they are superscalar processor cores. Furthermore, the MLCA does not require a program to be divided into fully parallel sections. It is able to exploit other types of parallelism, such as pipelinable loops.

• *Flexibility* – Because the MLCA is a template architecture, it can be easily tailored to any application’s performance, power and cost requirements. Such factors can be controlled by a system architect by varying the type and number of PUs, as well as the size and layout of the URF. Despite this flexibility the MLCA provides a unified programming model, so that applications written for a particular instance can be retargeted to another instance simply through recompilation.
Chapter 4

The Automatic Task Formation Framework

The simplicity of the MLCA’s programming model is one of its most compelling features. Parallel applications can be developed quickly and easily, as the hardware relieves the programmer of the burden of designing and implementing a correct and efficient synchronization scheme. As tasks are built the programmer must simply specify the data that is read and written in the task parameter lists; the hardware then uses this information to correctly synchronize tasks as necessary. The creation of task parameter lists is very similar to the process of creating function parameter lists in a standard sequential programming environment, so it comes naturally to most programmers. As an application is built the task parameter lists can be developed incrementally, with parameters being added as new data access requirements arise. The MOC further simplifies this process by automatically handling many of the complexities introduced by false sharing and unnecessary synchronization.

Unfortunately the process of porting an existing, sequential application to the MLCA’s programming model is not quite as simple. The code must first be refactored into tasks, then each task must be carefully analyzed to determine all of its data access requirements.
Each data location (ie. local or global variables, and heap-allocated memory) that is accessed by a task must be considered, and if a location is also accessed by other tasks then it must be allocated to the URF and added to task parameter lists. Furthermore, a control program must be developed that preserves the original application’s control flow.

This process is demonstrated in Figure 4.1. It is first determined that func2() should become a task. As a result, all code before and after the call to func2() must also either made into tasks or incorporated into the Sarek. Portions of main() and func1() must therefore also be turned into tasks. Several of the new tasks read and write to the variable i from the main function, so that variable is allocated to a URF register and added to task parameter lists accordingly. Maintenance of the original program’s control flow is complicated by the fact that Sarek only supports a subset of C operations. Tasks such as Eval_Parity() and Decrement() must therefore be introduced to evaluate conditions for loops and if statements. Sarek also does not support function calls, so control flow from multiple functions must be combined into a single control program. This can be seen in the example, as the while loop in main() and the if-else statement in func1() are effectively both inlined into the resulting Sarek.

Determination of task data access requirements can be significantly complicated by the presence of pointers, as they allow a single statement to access a different data location each time it is executed. If a task body uses pointers it is therefore necessary to determine all possible targets for those pointers and, if some of those targets are accessed by other tasks, then they must be added to the parameter list.

The process of task formation and code generation from existing sequential programs can be done manually, but it can be greatly simplified with appropriate compiler support. To this end we propose the Automatic Task Formation (ATF) Framework. It is a set of compiler tools that, combined with the MOC, provide a complete tool-chain to streamline the process of creating efficient translations of sequential applications to the MLCA’s programming model.
Figure 4.1: Porting a basic C program
Chapter 4. The Automatic Task Formation Framework

4.1 Architecture of the ATF Framework

The ATF framework consists of two main components: a task formation module and a code generation module. The task formation module uses a heuristic to identify potentially independent portions of an existing program, and to partition them into separate tasks. This provides the MLCA hardware with opportunities to dispatch tasks in parallel and, as such, an important consideration of the task formation module is task granularity. Large tasks may limit the MLCA’s ability to exploit parallelism by encapsulating independent sections in a single execution unit. Small tasks may also limit performance by introducing large amounts of task dispatch overhead and many inter-task dependencies. The task formation modules must therefore choose tasks of a sufficient granularity to balance these issues and maximize performance.

Once the task formation module has identified potential task boundaries within existing code, the code generation module uses this information to create new code that implements these tasks. It refactors the existing task bodies into new task bodies, and identifies any new inter-task data flow that this partitioning may have created. It then expresses the updated data and control flow in a new control program.

Figure 4.2 illustrates how the ATF components fit together, and how they can be
integrated with the MOC to form a full tool-chain. An existing MLCA program is given to the task formation component, which iteratively applies its heuristic to split tasks until a partition of appropriate granularity has been created. Newly-created task boundaries are expressed in the initial source by inserting pragma statements. The annotated source code is then passed to the code generation component, which creates a modified control program and task bodies. This new program can then be optimized by the MOC, and the resulting code can be compiled by GCC and the Sarek compiler to provide executable code.

The use of a pragma to denote task boundaries in the initial source code has a number of benefits. It enhances the modularity of the ATF framework by separating automatic task formation and task code generation. It also provides programmer control since pragmas can easily be manually inserted in or removed from the source code. Finally, the use of pragmas supports cross-platform development because pragmas are simply ignored by a compiler if it does not support them.

Despite the fact that the ATF framework requires an existing MLCA program as input, it is highly suitable for porting a sequential application (written in C, for example) to the MLCA. A trivial port, consisting of a single task whose body is the original program’s main function, can be created and used as input to the ATF framework.

The MOC has already been described in Section 3.3. Design considerations and a simple algorithm for the task formation component are presented in Chapter 5. Chapter 6 provides an in-depth discussion and design for the code generation component.
Chapter 5

A Task Formation Algorithm

In this chapter we present a heuristic algorithm to partition a program into tasks. Section 5.1 presents an overview of the algorithm’s goals, as well as some insights that influence its design. Section 5.2 describes a program annotation that the task formation algorithm uses to communicate the partition it creates to the later code generation phase. Finally, the task formation algorithm itself is formally specified in Section 5.3.

5.1 Algorithm Overview

The intent of this algorithm is to divide a program into tasks such that the MLCA architecture can execute the program as quickly as possible. This is primarily accomplished by identifying independent portions of the program and placing them in separate tasks. It is not sufficient, however, to simply maximize the amount of parallelism between tasks – consideration must also be given to the overheads (such as URF access and dispatch latency) introduced by partitioning the program. An ideal algorithm would minimize program execution time by balancing parallelism and overhead. Unfortunately, doing so has been proven to be an NP-complete problem so we focus on the development of an heuristic solution.

The MLCA is primarily intended for use in embedded multimedia applications. Through
examination of a variety of such applications we developed several insights that moti-
vated the design of the task formation algorithm. Multimedia applications tend to focus on processing streams of regularly-structured data units, such as blocks of pixels or audio samples. They are therefore often characterized by a single long-running outer loop, each iteration of which processes a single data unit. Furthermore, the processing of a single data unit generally breaks down into several distinct phases, such as downsampling, discrete cosine transformation and Huffman coding. There tends to be true data flow from earlier to later phases of processing for a single data unit, but there is often significant independence between different data units. In some applications data units are entirely independent of each other, and can therefore be processed in parallel. When this is not the case the dependencies between data units tend to exist only between corresponding processing phases, providing for the possibility of pipelined processing.

The algorithm is not intended to be treated as a black box, whose input is serial code and whose output is an MLCA program that exhibits significant parallelism. Rather, it is intended to be a transparent tool that can allow a programmer to rapidly port and tune an application for execution on the MLCA. It performs a source-to-source transformation in which both its input and output are MLCA programs, composed of a control program and a set of task bodies. It takes an iterative, top-down approach: it examines existing tasks and selectively splits them into smaller tasks. It can then be reapplied to the resulting tasks, causing them to be split further. The number of iterations in which the algorithm is applied is left to the discretion of the programmer, who can evaluate when the partition has reached a desired granularity. Furthermore, the programmer is free to modify the output of the algorithm before beginning a subsequent iteration, thereby providing additional feedback to tune the parallelization process.

It should be noted that the requirement of a MLCA program as input for the algorithm does not preclude its use in porting and parallelizing legacy applications written purely in languages such as C. It is trivial to turn such applications into a MLCA program by
void myfunc() {
  /* code 1 */
#pragma mlca Split
  /* code 2 */
}

(a) Original C
(b) Resulting Sarek

Figure 5.1: Basic split pragma usage

simply creating a single task whose body is the original program’s main function. Since
the resulting program contains only one task it will not exhibit any parallelism, but it
will be a suitable input for this algorithm.

5.2 The Split Pragma

The task formation algorithm is implemented as a source-to-source translation, in which
the original task bodies are annotated with newly identified task boundaries. To allow
this annotation, we have introduced a new C pragma to express explicit task boundaries.
It is:

#pragma mlca Split

The compiler places all code from the last boundary up to the pragma in a task, and all
code from the pragma to the next boundary in a second task. Figure 5.1 demonstrates
a basic use of the Split pragma. 5.1(a) shows the application’s original C code, and
5.1(b) shows the Sarek control program that would be generated.

Figure 5.1 also provides an example of implicit task boundaries. The split pragma
specifies the end of the first task and the beginning of the second task, however no
pragmas are necessary to specify the beginning of the first task and the end of the second
task. This is because function bodies are implicit task boundaries. That is, if the body
of a function contains any task boundaries then its beginning and end are also task
boundaries. The full list of C constructs that can be implicit task boundaries are:
void myfunc() {
  /* code 1 */
  while (cond) {
    /* code 2 */
#pragma mlca Split
    /* code 3 */
  }
  /* code 4 */
}

main() {
  myfunc_Task1(); // code 1
  cr1 = myfunc_Task2();
  while (cr1) {
    myfunc_Task3(); // code 2
    myfunc_Task4(); // code 3
    cr1 = myfunc_Task2();
  }
  myfunc_Task5(); // code 4
}

(a) Original C

(b) Resulting Sarek

Figure 5.2: Split pragma in a loop body

- Function bodies
- Loop bodies
- Conditional bodies
- Function calls (if the callee will be split)
- Return statements

These boundaries are made necessary by the Sarek programming model. Figure 5.2, which shows the effect of a split pragma in a loop body, demonstrates this fact. Since the loop is split it is moved from the C code into the Sarek code. A task body cannot span a Sarek loop boundary, so the start and end of the loop must be task boundaries. It is important to note that the C constructs listed above are not always treated as implicit task boundaries. They are only treated as such if they surround other task boundaries.

A more formal discussion of the Split pragma semantics and how they apply to the code generation process is presented in Section 6.1.

The use of a pragma to denote task boundaries has a number of benefits. These are:

- Modularity: automatic task formation and task code generation are logically distinct operations. Split pragmas provide a method for task boundaries to be communicated between the two phases, so they can be implemented as separate modules.
• *Programmer Control:* pragmas can easily be inserted in or removed from the source code by a programmer. This allows the programmer a higher level of control – they can modify the results from the automatic task formation pass to better suit their needs, or even perform all task formation manually – while still performing the more tedious code transformations automatically.

• *Cross-Platform Development:* pragmas are simply ignored by a compiler if it does not understand them. Thus the same code base can be compiled and run on traditional sequential architectures and on the MLCA.

### 5.3 The Algorithm

Figure 5.3 presents the task generation algorithm in pseudocode. Its entry point is the `Form_Tasks` function, whose input is an MLCA program (Sarek and corresponding task bodies) and whose output is a set of modified task bodies. The modified task bodies differ from the originals only in the addition of directives to specify new task boundaries, such as the `Split` pragmas described previously. These modified task bodies can then be processed by a code generator to produce fully modified Sarek and task bodies, which would be suitable input for further applications of the algorithm.

The algorithm considers each task individually, in a single pass. It traverses the statements in a task and inserts new boundaries in two situations: after loops and at certain function calls. These code features have been chosen as boundary points because they tend to be indicators of processing phases.

Loops generally account for a large portion of a program’s running time, so by inserting task boundaries after each loop the algorithm attempts to partition significant amounts of computation into separate tasks. When successful this provides tasks of a sufficiently coarse granularity (as they encapsulate at least one loop) to amortize the overhead of task creation, while still generating multiple tasks that can potentially be
// tunable parameters
int ST;    // function splitting threshold;
int AS;    // array access score
int LS;    // loop score
// end tunable parameters

Form_Tasks(in: Sarek_Program s, in: Task_Code c,
           out: Task_Code mod_c)
{
    mod_c = c;

    foreach Task t in s {
        Partition_Task(t, mod_c);
    }
}

Partition_Task(in: Task t,
               inout: Task_Code mod_c)
{
    t_c = Get_Task_Body(t, mod_c);

    foreach Statement s in t_c {
        if (s is a loop) {
            Insert_Directive_After(s, mod_c);
        } else if (s is a call to Function f) {
            if (Score(f) >= ST) {
                Insert_Directive_At_Start(f, mod_c);
            }
        }
    }
}

Score(in: Function f)
{
    int score = 0;

    for each Statement s in f {
        if (s is a call to Function g) {
            score += Score(g);
        } else if (s is an access to Array a && !Is_Read_Only(a)) {
            score += AS;
        } else if (s is a loop) {
            score += LS;
        }
    }

    return score;
}

Figure 5.3: Task generation algorithm
dispatched in a fully parallel, or at least pipelined manner.

The rationale for using function calls as task boundaries is similar. Programmers will usually divide logically distinct computations into separate functions, so it is reasonable to assume that different processing phases will each be encapsulated in their own function. By inserting a boundary directive at the start of a function we ensure that the body of the function will become a task. It is important to note that functions are not split recursively – when a boundary is added to the body of a callee, the body of that function is not considered for the addition of further task boundaries, even if it contains loops and function calls. This is because of the iterative nature of the algorithm. If it is decided that the granularity of the partition being generated is not sufficiently fine to expose the parallelism available in the program, the task formation algorithm can be applied again. On this next application the body of the callee will be considered, as it will have been promoted to a task.

Using function calls as task boundaries serves to separate logically distinct operations into separate tasks but, if applied indiscriminately, would introduce unacceptable overheads. Many functions perform a very small amount of work, such as setting a flag or performing a minor computation. Promoting such functions to tasks can provide further opportunities for parallelism, but the overhead introduced is quite large relative to the amount of work performed in the function bodies. Thus it is rarely beneficial to promote small functions to tasks. Indeed it is often detrimental to overall program performance since small functions are generally executed frequently, so turning them into tasks greatly increases the proportion of overhead to useful computation in the program as a whole.

To prevent the creation of large numbers of excessively fine-grained tasks, the algorithm incorporates a simple scoring system to determine whether or not it should insert a task boundary for a given function call. It calculates a score, $S(f)$ for each function $f$ according to the following formula:

$$S(f) = \sum_{c \in FC(f)} S(Callee(c)) + (LS \times L(f)) + (AS \times A(f))$$
FC(f) is the set of all function calls in f, L(f) is the number of loops in f and A(f) is the number of array access statements in f. A(f) is calculated to exclude accesses to arrays whose contents remain constant at run-time (read-only arrays). LS and AS are loop and array scaling factors, and are tunable parameters. If a function’s score is greater than a certain threshold, ST, a task boundary is added. Otherwise the call is ignored. This has the effect of creating new tasks out of functions that perform a significant amount of computation, while leaving smaller function calls as part of larger tasks.

The reasoning behind the incorporation of callee scores and the number of loops into a function’s score is straightforward. Both are fairly direct measures of the amount of computation performed in a function, so they justifiably increase its score. The reasoning behind the use of the number of array accesses is perhaps more subtle. Previous work has shown that the bulk of the important data flow in multimedia programs is accomplished through arrays [12]. This is reflected in the importance of optimizations such as array privatization and renaming in MLCA programs. Adding task boundaries in functions that access arrays ensures that this data flow will be spread across task boundaries, which creates the potential for such optimizations. This ultimately leads to a program that can be effectively parallelized by the MLCA.

A notable feature in the calculation of A(f) is that it excludes accesses to read-only arrays. This is because access to such arrays is typically sparse, with only a few elements being used at one time. For example, such arrays might be tables of scaling factors indexed by a value in a frame header. A single value would be read from the array at the start of processing for each frame. This is not a sufficient amount of computation to justify encapsulation in a task, so it is ignored. In cases where large portions of a read-only array are accessed at one time (thereby justifying the creation of a task), there are almost always corresponding accesses to other arrays whose values are not run-time constants. Accesses to those arrays would be included in A(f), so the computation would
increase the function’s score.

Figure 5.4 shows two iterations of this algorithm applied to a small C program. We assume that initially `main()` is the only task. The first task formation iteration therefore examines its body and adds a `Split` pragma to the beginning of `func1()`. The algorithm only examines one call-level deep inside existing tasks, so it does not consider `func2()` during this iteration. At the start of the second iteration, however, `func1()` has been promoted to a task and its body is therefore analyzed. As a result `Split` pragmas are added at the beginning of `func2()` and after its first loop. The resulting Sarek that would be generated after the second iteration is also shown. Note that all three tasks can be dispatched in parallel.

This algorithm is very simple, and it lacks the sophistication of many other automatic task formation techniques [26, 15, 29]. Despite this simplicity the algorithm provides good performance when applied to a range of multimedia applications, as is shown in the evaluation presented in Chapter 7. It is nonetheless intended primarily as a proof-of-concept for the ATF framework, and leaves significant room for improvement. Possible directions for future work on this algorithm are discussed in Section 9.1.
int main(int argc, char **argv) {
    func1();
    return 0;
}

void func1() {
    printf("This is func1\n");
    func2();
}

void func2() {
    int i, j;
    int a=1, b=1;

    printf("This is func2\n");
    for (i = 0; i < 10; i++) {
        a *= 2;
    }
    for (j = 10; j; j--) {
        b *= 3;
    }
}

cr_t main() {
    // prints "This is func1"
    func1_Task1();
    // prints "This is func2"
    // also calculates 2^10
    func2_Task1();
    // calculates 3^10
    func2_Task2();
}

Figure 5.4: An example of automatic task generation
Chapter 6

Code Translation and Generation

The fundamental job of the code generation component is to translate between the standard sequential C programming model and the MLCA’s hybrid C/Sarek programming model. This involves a partitioning of the program functionality – some of the control and data flow of the original program must be expressed in the Sarek control program, and the rest must be preserved in the C code that implements the task bodies. This chapter explores these considerations in detail, and presents the algorithms used by the ATF framework during code generation. Section 6.1 describes the semantics of a C pragma that delineates task boundaries in the original source code. Section 6.2 describes the procedure by which task bodies are generated. Section 6.3 sets forth the methods by which inter-task data flow is determined. Section 6.4 describes the special data-flow handling necessitated by the use of pointers. Section 6.5 describes the techniques by which the original program’s control flow is preserved. Section 6.6 describes a number of minor data flow optimizations.

6.1 Task Boundary Determination

The code generator begins by determining task boundaries within the original C code. These boundaries fall into two categories: explicit and implicit task boundaries. Explicit
boundaries are those that are mandated by whomever is porting the application to the
MLCA – either an automatic task formation utility or a programmer. Implicit boundaries
are those that are mandated by the MLCA programming model.

6.1.1 Algorithm Overview

We now present an algorithm to identify explicit and implicit task boundaries in a pro-
gram that has been annotated with \texttt{Split} pragmas. This algorithm takes an MLCA
program as input, and produces a modified MLCA program as output. The fact that
it requires an already-ported MLCA application as input does not limit its usefulness,
however, as the process of porting a pure C application is trivial\footnote{The program’s main function is made into a task body, and a control program is generated that consists of a single call to that task.}

Figure 6.1 shows the pseudocode for the algorithm. It examines each task that is
called by the control program, looking for task boundaries enclosed in the task’s body.
If boundaries are found then the task body is split accordingly, and the task call in the
control program is replaced with calls to the newly created tasks. If any tasks were
split, then the process is repeated so that split points within newly created tasks will be

```c
Split_Tasks(in: Sarek_Program s, in: C_Code c,
            out: Sarek_Program mod_s, out: C_Code mod_c) {
  mod_s = s;
  mod_c = c;
  change = true;
  while (change) {
    change = false;
    foreach Task t in mod_s {
      if (t contains task boundaries) {
        Split(t, mod_s, mod_c);
        change = true;
      }
    }
  }
}
```

Figure 6.1: Task splitting algorithm
handled.

### 6.1.2 Control Flow Propagation

Task boundaries may occur within C control flow constructs, and in such situations the control flow must be moved into the control program. Figure 5.2 has already demonstrated how a task boundary within a *while* loop is handled. *Do-while* loops are handled similarly, however the initial call to the conditional-evaluation task is replaced with a simple assignment of the control register to ensure that the first iteration of the loop always executes.

Figure 6.2 on the next page demonstrates how all other C structured control flow constructs are converted to Sarek. 6.2(a) and 6.2(b) show how a C *if* statement is replaced by a Sarek *if* statement. The effect is identical for split pragmas in the else clause. 6.2(c) and 6.2(d) show how a *for* loop is handled. Sarek does not support *for* loops, so the loop must be converted to a *while* loop. This is accomplished by creating tasks to execute the initialization and increment statements of the loop, and inserting calls to them at the appropriate points in the Sarek. 6.2(e) and 6.2(f) show how function calls are handled. Recall that they are only considered to be task boundaries if the callee contains split points. Therefore the call to *foo()* forms an implicit task boundary, but the call to *bar()* does not. It is also worth noting that the call to *foo()* does not get encapsulated in a task; rather it is replaced by calls to the tasks that are split from the callee’s body. This process is somewhat analogous to function inlining, as the body of *foo()* has been incorporated into the control program.

The C language also contains support for unstructured control flow through the use of labels and gotos. Sarek does not contain similar constructs (though HASM does). It is therefore not possible to express unstructured control flow if it spans task boundaries, and this algorithm makes no attempt to do so.
void myfunc() {
    /* code 1 */
    if (cond) {
        /* code 2 */
        #pragma mlca Split
        /* code 3 */
    }
    else {
        /* code 4 */
    }
    /* code 5 */
}

main() {
    myfunc_Task1(); // code 1
    cr1 = myfunc_Task2();
    if (cr1) {
        myfunc_Task3(); // code 2
        myfunc_Task4(); // code 3
    }
    else {
        myfunc_Task5(); // code 4
    }
    myfunc_Task6(); // code 5
}

void myfunc() {
    /* code 1 */
    for (init; cond; incr) {
        /* code 2 */
        #pragma mlca Split
        /* code 3 */
    }
    /* code 4 */
}

main() {
    myfunc_Task1(); // code 1
    myfunc_Task2(); // for loop init
    cr1 = myfunc_Task3(); // for loop cond
    while (cr1) {
        myfunc_Task4(); // code 2
        myfunc_Task5(); // code 3
        myfunc_Task6(); // for loop incr
        cr1 = myfunc_Task3();
    }
    myfunc_Task7(); // code 4
}

void myfunc() {
    /* code 1 */
    foo(); // has task bounds
    /* code 2 */
    bar(); // no task bounds
}

main() {
    myfunc_Task1(); // code 1
    foo_Task1();
    foo_Task2();
    myfunc_Task2(); // code 2 and bar()
}

Figure 6.2: Splitting C control flow
6.1.3 Ignored Split Pragmas

When the algorithm examines a task body for split points, its analysis only looks one call-level deep. This introduces the possibility of split pragmas that will be ignored. Figure 6.3 demonstrates a situation in which this would happen. `func_withPragma()` contains a split pragma, however it is two call-levels deep in the body of `mytask()`. So, the algorithm would not split `mytask()`.

This aspect of the algorithm is not fundamentally necessary – it would be trivial to modify it so that the algorithm looks arbitrarily deep in the call tree – but it does simplify implementation of the algorithm. We view this as a reasonable design trade-off as a compiler could output a warning if a program contains ignored split pragmas, and additional split pragmas could be added to solve the problem.

6.2 Task Body Generation

Once task boundaries have been found in the original C code the new task bodies must be created. The Sarek programming model requires that each task be implemented as a C function, so functions must be created to implement these new tasks. This is done through a process called outlining.

Outlining takes a portion of function’s body and encapsulates that portion in an
independent function. In the context of a normal C program, that portion of code is then replaced with a call to the newly created function. In the context of the MLCA the call is added in the Sarek program instead. Outlining is the logical opposite of function inlining, in which function calls are replaced by the bodies of the functions themselves.

The ATF framework builds upon an implementation of the outlining transformation in the open-source ORC compiler to create new task bodies. This implementation was provided by Peng Zhao of the University of Alberta [33]. The analyses and transformations described in the following sections were added to this implementation to adapt it to the MLCA’s programming model, and to fit the ATF framework’s needs.

6.3 Task Data Flow

In all but the most simple programs there is some degree of data flow across task boundaries. In the pure C code that is being split this data flow is largely implicit – some statements write to variables or other memory locations, and other statements read from variables or memory. In contrast, Sarek requires that data flow be declared explicitly in the form of task parameters. Each task call must specify which URF registers it reads and which registers it writes. When splitting a stretch of C code into multiple tasks the compiler must analyze that code to determine which data values span task boundaries. It must then allocate the memory locations that store those values to URF registers and add those registers as input and output arguments to the new tasks as appropriate.

In C, data locations can be divided into two categories: scalars and aggregates. Scalars are variables of the basic data types: integers, floating-points, characters and pointers. Access to them is atomic in the sense that they can be read or written in a single operation, and all reads or writes affect the entire scalar. We assume that all C scalars fit in a single URF register, therefore they may be allocated directly to the URF and their use in data flow analysis is straightforward.
Aggregates are groups of scalars: arrays and structures. Access to them is not atomic, as each element must be accessed individually. We assume that they do not fit in a single URF register, so they cannot be passed between tasks through the URF. Instead they must be allocated in shared memory, and pointers to this memory must be passed in the URF.\footnote{Strictly speaking it may be possible to fit aggregates into a single URF register, as some instances of the MLCA may include long or variable-length URF registers. In such a situation aggregate data flow becomes identical to scalar data flow, so it will not be discussed further.}

Pointers pose a particular problem during task creation since they bypass the synchronization model that the URF provides. They do so in two ways. Firstly, a C pointer cannot point at a URF register—it may only point at task-local or shared memory locations, upon which the MLCA provides no synchronization. We call this the pointer naming problem, and explore it in further detail in Section \[6.4.1\]. Secondly, pointers perform indirect access to data locations. Thus a single statement that reads or writes data through a pointer may target a different location each time it is executed. This phenomenon, known as aliasing, and its implications upon task formation are detailed in Section \[6.4.3\].

This section describes the data flow algorithms that the ATF code generator uses to determine task inputs and outputs. The special considerations that pointers cause, and the way in which the ATF code generator incorporates them into its basic data flow analyses are detailed in Section \[6.4\].

The design of the following algorithms was heavily influenced by the fact that the ATF code generator sits in the middle of the overall MLCA tool chain. As a result their primary focus is on ensuring program correctness rather than maximizing parallelism. In some cases, particularly with regards to tasks that access aggregate data, they may over-constrain a program by introducing conservative task synchronization. This does not pose a major performance problem as the MOC is a downstream link in the tool chain. Its optimizations successfully remove most unnecessary synchronization, allowing
code produced by the ATF code generator to ultimately exhibit good of parallelism.

6.3.1 Program Analysis

At the time of analysis each task is just a stretch of code within a function that is being split. The compiler must examine that code to determine what memory locations it accesses, as well as whether access to each location is read-only, write-only or read/write. This allows it to decide which URF registers a task needs to access, and whether they should be declared as \texttt{in}, \texttt{out} or both.

Not all memory accesses within a task require a corresponding URF access. If a task reads a variable but always writes a value to that variable first, then that variable’s URF register does not need to be a task input. This is demonstrated in Figure 6.4. Function \texttt{myfunc} will be split into two tasks; the first does not access any variables, and the second accesses all three variables. This second task writes to \texttt{a} and \texttt{b}, so they must be task outputs. It reads all three variables, however \texttt{a} is always written before it is read. For this reason, \texttt{a} does not need to be an input. \texttt{b}, on the other hand, is only sometimes written before it is read. For this reason it must be a task input.

As demonstrated above, the set of memory locations that a task must declare as inputs is a subset of all memory locations that it reads. Specifically, that subset is all targets of \textit{upward-exposed reads}. An upward-exposed read is an operation that reads a value that may have been defined before the start of the task. To find the set of upward-exposed reads in a task the compiler must perform a standard any-path, backward data flow analysis on that task’s body \cite{27}.

To perform this analysis the task body is divided into basic blocks, and the basic blocks are connected to form a control-flow graph. Each basic block has several associated sets of variables:
void myfunc(int my_arg) {
    int a, b, c;
    printf("start of myfunc\n");
#pragma mlca Split
    if (my_arg) {
        a = 10;
        b = 5;
    } else {
        a = 12;
    }
    printf("%d %d %d\n", a, b, c);
}

main() {
    reg_t a, b, c;
    myfunc_Task1();
    myfunc_Task2(in b, in c,
                 out a, out b);
}

(a) Original C

(b) Resulting Sarek

(c) Control Flow Graph

Figure 6.4: Scalar data flow
\( \text{Gen} = \{\text{all variables that are read, but not previously written, in the block}\} \)

\( \text{Kill} = \{\text{all variables that are written in the block}\} \)

\( \text{In} = \{\text{all upward exposed reads at the start of the block}\} \)

\( \text{Out} = \{\text{all upward exposed reads at the end of the block}\} \)

The Gen and Kill sets are initialized according to the contents of each basic block. The In and Out sets can be initialized to \( \phi \), though initializing In to Gen can help the analysis converge more quickly. Backward analysis can then be performed using the following data flow equations:

\[
\text{Out}(blk) = \bigcup_{s \in \text{Succ}(blk)} \text{In}(s)
\]

\[
\text{In}(blk) = \text{Gen}(blk) \cup [\text{Out}(blk) - \text{Kill}(blk)]
\]

Figure 6.4(c) shows the control-flow graph for the second task in Figure 6.4(a). Block 1 corresponds to the if clause, block 2 to the else clause, and block 3 is the remainder of the task. The variable sets for each block are also shown with contents as they would be after analysis is complete.

The results of the analysis can be used directly to determine a task’s input and output registers:

\[
\text{Input Registers} = \text{Out}(\text{top})
\]

\[
\text{Output Registers} = \bigcup_{b \in \text{CFG}} \text{Kill}(b)
\]

In other words, a task must input each variable that is involved in an upward-exposed read (with respect to the start of the task), and it must output all variables that it writes. This algorithm guarantees that the input set will be minimal – a task will read all registers required for correctness and no more. In contrast, the output set is overly conservative. A register needs to be output only if another task will use it as input, however this algorithm marks all variables that are written as output regardless of whether or not they are subsequently used. Doing so will not directly detract from the parallelism of
the resulting program, however it can indirectly cause some undesirable performance effects. These effects, as well as a simple downstream optimization to eliminate them, are described in Section 6.6.2.

6.3.2 Conditional URF Writes

Sarek requires that tasks have static argument lists – they explicitly declare all task inputs and outputs. It is expected that the task body will contain exactly one call to the URF access API for each argument listed. This does not cause any difficulties for URF reads, but it does require extra consideration for URF writes. Care must be taken to ensure that a value has actually been produced in the task body before a call to the URF write function occurs.

Figure 6.5 demonstrates this problem. Figure 6.5(a) shows a Sarek program, and Figure 6.5(b) shows an initial attempt at creating corresponding task bodies. Task cond_write sometimes, but not always, produces a value that is written to val, yet the call to writearg happens unconditionally. In situations where a value is not produced by cond_write an uninitialized value will be written into the URF for val, and the output of task read will be undefined.

There are two possible solutions to this problem. The first is to always read URF registers that are involved in conditional writes. Doing so guarantees that if a value is not produced the old value will be correctly propagated. The second is to guard the writeArg statements in the task’s body such that they only execute when a value is produced. This technique is demonstrated in Figure 6.5(c).

Both techniques preserve program correctness in the presence of conditional writes, but each has different benefits and drawbacks. The first technique can, in situations where tasks perform conditional write-only access to a register, unnecessarily limit the parallelism in a program. Because access is write-only, such tasks should be able to run in parallel with prior tasks that also write to the same register (assuming register renaming
main() {
    reg_t cond, val;

    write(out val);
    cond_write(in cond, out val);
    read(in val);
}

(a) Sarek

void write() {
    writeArg(0, 0);
}

void cond_write() {
    int cond = readArg(0);
    int val;
    if (cond) {
        val = 10;
    }
    writeArg(0, val);
}

void read() {
    int val = readArg(0);
    printf("Value is %d ", val);
}

(b) Incorrect Task Body

void write() {
    writeArg(0, 0);
}

void cond_write() {
    int cond = readArg(0);
    int val;
    int write_flag1 = 0;
    if (cond) {
        val = 10;
        write_flag1 = 1;
    }
    if (write_flag1)
        writeArg(0, val);
}

void read() {
    int val = readArg(0);
    printf("Value is %d ", val);
}

(c) Correct Task Body

Figure 6.5: Conditional URF Writes
main() {
  reg_t myreg;
  write(out myreg);
  cond_write(out myreg);
  read(in myreg);
}

(a) Original Sarek

main() {
  reg_t myreg;
  write(out r1);
  cond_write(out r2);
  read(in r2);
}

(b) After Register Renaming

Figure 6.6: Conditional Writes and Register Renaming

is performed). If the register is added as an input, the task is forced to wait for all prior
writes to complete before it can be dispatched. The central benefit of this technique,
however, is that it does not require any additional hardware support.

The second technique does not add any additional register reads, so it does not reduce
the available parallelism in a program, but it does require additional hardware support.
Nothing special needs to be done in situations where a value is produced, but care must
be taken when a task completes without producing a value for one or more of its output
registers. If register renaming has been performed then the value must somehow be
propagated from the original register to the renamed register before subsequent tasks
can read the renamed register.

Figure 6.6 demonstrates this problem. An original Sarek program is shown in Figure
6.6(a) and the same program is shown after register renaming has been performed in
6.6(b). The logical register myreg has been renamed to two different physical registers,
allowing tasks write and cond_write to execute in parallel. If task cond_write produces
a value then the program will work as expected, but if it does not then r2 will not be
initialized and task read will read an incorrect value. Program semantics would require
that it read the value stored in r1 instead.

In order to allow for conditional writes to task arguments, the renaming hardware
must be able to recognize situations at run-time in which a task has not produced values
for all of its output arguments. It can then handle the situation in one of two ways.
The first is to wait until a task completes before propagating the results of its output register renaming to future tasks. In the example in Figure 6.6 this would mean waiting to perform renaming on read’s input arguments until cond_write has completed. If a value is produced then read should use r2 as an argument, otherwise it should use r1.

The second option is to copy the value from the old physical register to the new physical register when a task does not produce a value for one of its output arguments. In the context of Figure 6.6 this would mean copying the value from r1 into r2 if cond_write does not produce a value. Extra care must be taken before performing this copy, as it is likely that write and cond_write will execute in parallel. If cond_write completes before write has written a value to r1, then it must stall until write completes before performing its copy.

The ATF framework was designed to target a simulated instance of the MLCA (described in further detail in Section 7.2.2). The simulator was modified to allow conditional writes by copying the original value to the renamed register when a task does not produce a value. As such, no further compiler support for condition writes was necessary in the ATF framework.

### 6.4 Pointer Support

Pointers allow indirect access to data locations, and this poses a number of significant problems for a program that will run on the MLCA. These problems all stem from the fact that pure C uses a unified memory model as opposed to the MLCA’s hybrid Sarek/C programming model, which uses a segmented memory model. MLCA programs can access data in two separate locations – memory or the URF\(^3\) – and access to each of those locations is performed differently. Access to memory is performed implicitly by

\(^3\)Actually, MLCA programs can access data in three different locations: processor-local memory, shared memory and the URF. For the purposes of this discussion, though, local and shared memory can be considered together.
standard C operations, and access to the URF is performed explicitly through calls to a C API.

Pointers, by nature, depend upon C’s unified memory model. They are simply data locations that store memory addresses of other data locations. As such, they cannot directly reference locations in the URF. The URF, however, is critical to the correct operation of a program on the MLCA. It is intended to be the main conduit for data flow between tasks, and it is the only means for synchronization between tasks. If a variable is allocated to a URF register, then any access to that variable by a pointer dereference will bypass the URF entirely.

6.4.1 The Pointer Naming Problem

The fact that pointer usage bypasses the URF manifests itself as several separate but related problems. The first is a direct consequence of the allocation of variables to URF registers. The C programming model assumes a one-to-one mapping between variables and memory locations; that is, each variable exists at exactly one memory address. This
assumption is necessary for the maintenance of data coherence. When a program is ported to the MLCA, some of its variables are allocated to URF registers. From a coherence standpoint, those registers replace memory locations in the one-to-one mapping: they become the only location in which a variable’s data should be stored. When all accesses to variables within a task body are direct, this is not a problem: they are performed on task-local copies of the variable that are synchronized with the URF as necessary. Indirect access to variables, via pointers, is more problematic. In pure C pointers store the address of a variable’s memory location. In the MLCA programming model, that memory location will no longer have an address if it has been allocated to the URF.

Figure 6.7 illustrates this dilemma. The original program contains a global variable \( a \), and a function that will be split into two tasks. The behaviour of the original code is straightforward. When the program is ported to the MLCA, however, \( a \) is allocated to a URF register. This register replaces the global memory location: all tasks work on local copies of the variable, whose values they synchronize with the URF as appropriate. The value of \( ptr \) is therefore set to the address of \( myfunc_{-}Task1 \)’s local copy. This value is passed to \( myfunc_{-}Task2 \), which subsequently dereferences it. This dereference is a program error, as its target is a stack variable that almost certainly no longer exists.

At its root this is a naming problem. Pure C uses a variable’s address as the unique identifier, or “name”. Statements can then access variables indirectly, by specifying the name (address) of the variable they wish to access. The MLCA programming model also uses memory addresses as the name for variables that are not allocated to the URF. Variables that are allocated to the URF, in contrast, use their URF register number as their name. This naming scheme is incompatible with C’s pointer semantics.

### 6.4.2 Globalization

A variety of possible solutions to the naming problem exist. The simplest solution, which is used in the ATF code generator, is to avoid it altogether: if a variable may be the
target of a pointer dereference at any point in the program then the compiler may choose not to allocate it to a URF register. This directly solves the naming problem, as the variable will be allocated in memory and can therefore be referenced via pointers. It does present additional difficulties if the variable must be accessed by multiple tasks, however. As has already been demonstrated in Figure 6.7, the variable cannot be allocated in function-local (and therefore task-local) memory. It must be allocated in global memory. If a variable is already in global memory then no additional work is necessary. If it is in function-local memory, then it must be promoted to global memory and all direct accesses to it must be modified to reflect this change.

This globalization of variables solves both the naming problem and the inter-task data sharing problem, but there is one additional pitfall. Figure 6.8 shows the Sarek and C for a revised MLCA port of the application shown in Figure 6.7. Since \( a \) is already in global memory, and it is not allocated to a URF register, no code modifications are necessary. Unfortunately, the output of this program is unpredictable due to a race condition. Since \texttt{myfunc_Task3} has no arguments the MLCA’s out-of-order execution policies allow it to be executed at any time. If it happens to run after \texttt{myfunc_Task2} then the output will
be as expected, but there is no guarantee of that ordering. If it happens to execute before (or concurrently with) `myfunc_Task2` then the output will be incorrect.

Since \( \mathbf{a} \) has not been allocated to the URF, the MLCA’s scheduling facilities have no knowledge of it. They can therefore do nothing to avoid race conditions like the above. The solution is to allocate a register in the URF to act as a *synchronization proxy* for \( \mathbf{a} \). This register is not used for actual data storage – the naming problem prevents its usefulness for that purpose – but it is included as both an \texttt{in} and \texttt{out} argument for all tasks that access \( \mathbf{a} \). This introduces artificial flow dependencies between all tasks that access \( \mathbf{a} \), therefore causing all such tasks to execute in the original program order.

The addition of synchronization proxies is not limited to statically-allocated global variables. Any memory location that is shared between multiple tasks requires a proxy. This includes all dynamically allocated memory, as the heap is assumed to be in global memory.

The compiler, when generating tasks, must add these proxy URF accesses to ensure the correctness of the resulting program. It must therefore analyze the body of each task to find all global locations that are accessed. When access to a global location is performed directly this is trivial, however indirect access to global locations requires more sophisticated analysis.

### 6.4.3 Points-To Analysis

A statement that performs an indirect memory access has a single target location at runtime, however at compile time it may have many possible targets. The compiler must determine all possible targets for an indirect memory access so that use of a synchronization proxy can be added for each target that is in global memory.

Compile-time determination of all possible pointer targets is called *points-to analysis*. Points-to analysis algorithms present a large design space, and many different algorithms exist that make differing tradeoffs between accuracy and time complexity. Section 2.4
Chapter 6. Code Translation and Generation

presents a general description of points-to analysis.

For this work we have chosen to use Steensgaard’s Points-to Analysis in Almost Linear Time algorithm [30], which was already implemented in ORC. The details of this algorithm have been described in Section 2.4.1. The output of this algorithm can be used by the compiler to generate synchronization proxies in one of two ways.

The first is to allocate a single synchronization proxy for each alias class. This will guarantee program correctness, and it will do so with the introduction of a minimal number of synchronization proxies. To use this method the compiler must simply find all alias classes that are accessed in the body of a task, and add the corresponding proxies to that task’s argument list.

The second way is to allocate a single synchronization proxy for each memory location. This requires the use of a larger number of URF registers as synchronization proxies, but it also guarantees program correctness. To use this method the compiler must do some additional work: it must determine which variables are in the same alias class. It can do so by iterating across the entire program and building a set of variables for each alias class. Once this is known, the compiler can analyze each task. For each indirect memory access, it must add access to the synchronization proxies for all possible global targets.

Both methods provide ideal parallelism: they add access to synchronization proxies (and therefore task serialization) only where necessary to ensure program correctness. They do, however, have respective pros and cons. The first method adds a minimal number of proxies, therefore keeping URF register pressure minimal. The second method uses more registers, and could therefore hurt performance as a result of additional register pressure. It does, however, have several significant benefits:

• **Human-readability:** a programmer can directly determine all variables that a task accesses by examining its argument list (assuming that the proxies are given helpful

---

4Ideal, in this context, is with respect to the given alias information. If that alias information is overly conservative then the resulting program parallelism will be sub-optimal.
names). This would be much more difficult if the argument list just included the alias classes that are accessed.

- **Granularity**: Alias-analysis algorithms are generally overly conservative, therefore causing the addition of unnecessary synchronization proxies. If a programmer can determine that a specific task does not in fact access a particular memory location, that location’s proxy can be removed from the task’s argument list.

We have chosen to implement the second method. The ATF framework is intended as a transparent programming tool, so the benefits of this method to the programmer outweigh the overhead of its additional register pressure. Though we have not performed an extensive evaluation of this tradeoff, the experimental results presented in Chapter 7 support this assertion.

### 6.4.4 Context-Sensitive Points-To Analysis

The Steensgaard points-to analysis algorithm used in this work is context-insensitive, which means that if a function is called from multiple sites then the analysis will provide a single solution that covers all call-sites. This solution is likely a superset of the solution for any given call-site.

Figure 6.9(a) provides an example of such a situation: `myfunc` is called twice, each time with a pointer to a different variable. When it dereferences its pointer argument, the alias analysis algorithm must assign that statement to a single alias class. This class must include both `a` and `b`. 6.9(b) shows the resulting Sarek. Both variables’ synchronization proxies are included for both calls, despite the fact that each call only needs to synchronize on one of the variables.

A possible solution to this problem would be to use a more precise, context-sensitive algorithm. Such algorithms do exist, however they are very computationally intensive [19]. As an alternative, Whaley and Lam suggest a method to emulate context-sensitivity...
void main() {
    int a, b;
    myfunc(&a);
    myfunc(&b);
}

void myfunc(int *my_arg) {
    #pragma mlca Split
    printf("%d", *my_arg);    // AC 1
}

main() {
  reg_t main_a, main_b;
  myfunc_Task1(in main_a, in main_b,
               out main_a, out main_b);
  myfunc_Task1(in main_a, in main_b,
               out main_a, out main_b);
}

void myfunc_clone1(int *my_arg) {
    #pragma mlca Split
    printf("%d", *my_arg);    // AC 1
}

void myfunc_clone2(int *my_arg) {
    #pragma mlca Split
    printf("%d", *my_arg);    // AC 2
}

main() {
  reg_t main_a, main_b;
  myfunc_clone1_Task1(in main_a,
                      out main_a);
  myfunc_clone2_Task1(in main_b,
                      out main_b);
}

Figure 6.9: Context-Insensitive Alias Analysis

Figure 6.10: Cloning and Alias Analysis
under context-insensitive algorithms \[32\]. If a function has multiple call-sites, then a clone of that function is created for each site. Then context-insensitive analysis is performed. Since each function now has only one call-site the result is identical to that of context-sensitive alias analysis. Figure 6.10 shows how cloning can be applied to the previous example, and the resulting Sarek. Note that the addition of context sensitivity has significantly increased the parallelism of the program: there are no longer any data dependencies between the two tasks.

It is important to note that, if a function body contains function calls, creating a clone of that function will introduce new duplicate call-sites. A single application of the cloning transformation is therefore not always sufficient to eliminate all call-site duplication. This limitation can be solved by simply applying the cloning transformation iteratively until no duplicate call-sites remain\[^5\]. The problem with this approach is that it can potentially cause exponential code growth, which in many cases results in an unpractically large program size. The ATF code generator thus performs iterative cloning only to a user-specified cloning depth. A default cloning depth of 1 is used, as a single iteration of cloning is generally sufficient to enable precise alias analysis. When this is not sufficient the user can trade code size for precision by increasing the cloning depth.

\[6.5\] Task Control Flow

Sarek provides analogues for most C structured control flow statements. When a C function is being split into tasks, it is generally trivial to incorporate its control flow into the Sarek program: simple code-generation templates to do so are presented in Section 6.1.2. However, certain C control flow statements present some additional issues that the code generator must consider. These statements, and their attendant considerations, are described in this section.

\[^5\]Care must be taken to recognize recursion when performing iterative cloning, as the call-graph cycles that it introduces would prevent the algorithm from converging.
6.5.1 Function Parameters

When a function is split into tasks its call-sites are effectively incorporated into the Sarek code. This is mostly an intuitive process, as task calls in Sarek largely resemble function calls in C. Some extra effort is necessary, however, to support C’s parameter passing conventions.

C uses call-by-value semantics for function calls. The caller supplies an expression for each parameter. These expressions are evaluated, and their results are copied into the formal parameters of the callee. As demonstrated in Figure 6.11, these semantics can be only partially mapped to Sarek. The resulting Sarek program contains a call to a single task, myfunc_Task1. The mapping between expressions in main and formal parameters in myfunc is performed by providing the necessary URF registers in myfunc_Task1’s argument list. The result is incorrect, however, as the addition of 5 to b has been lost. Sarek does not have any facilities to do arithmetic on URF registers, so that expression can not be included in myfunc_task1’s parameter list.

One possible solution is to generate a parameter initialization task that is inserted in the Sarek before each newly incorporated call-site. The body of this task maps caller
main() {
  reg_t main_a, main_b;
  reg_t myfunc_p1, myfunc_p2;
  myfunc_InitTask1(in main_a, in main_b,
                   out myfunc_p1, out myfunc_p2);
  myfunc_Task1(in myfunc_p1, in myfunc_p2);
}

void myfunc_InitTask1() {
  int main_a = readArg(0);
  int main_b = readArg(1);
  writeArg(0, main_a);
  writeArg(1, main_b + 5);
}

void myfunc_Task1() {
  int p1 = readArg(0);
  int p2 = readArg(1);
  printf("params: %d %d ", p1, p2);
}

Figure 6.12: Parameter Initialization Tasks

expressions to callee parameters, performing expression evaluation as necessary. Figure 6.12 shows how the introduction of a parameter evaluation task solves the problem in the previous example.

Parameter initialization tasks solve the problem of mapping expressions to parameters, however they introduce extra task scheduling, startup and shutdown overhead. This overhead can be eliminated by merging them with either the task directly preceding the call-site (which is derived from the caller) or the task directly after the call-site (which is derived from the callee). In either case the merging procedure, and the resulting performance, is largely identical, so we only discuss the latter. This is the technique implemented in the ATF code generator.

Figure 6.13 demonstrates how the parameter initialization task can be merged into the first task of the callee. That task reads the necessary variables from the caller, then performs any necessary calculation and assignment to the formal parameters. It
main() {
  reg_t main_a, main_b;
  myfunc_Task1(in main_a, in main_b);
}

void myfunc_Task1() {
  int main_a = readArg(0);
  int main_b = readArg(1);
  int p1 = main_a;
  int p2 = main_b + 5;
  printf("params: \%d \%d ", p1, p2);
}

(a) Sarek

(b) Task Bodies

Figure 6.13: Merging Parameter Initialization Tasks

is important to note that, since the initialization task being merged is specific to the call-site, care must be taken if the callee has multiple call-sites. A separate merged task must be created for each call-site.

6.5.2 Return Statements

Return statements in the original C code present two problems. First, Sarek tasks can not return data values; they can only return values that are stored to control registers. If a non-void C function is split into tasks, some other mechanism must be used to communicate its return value to the caller. Second, a function may contain return statements at any point in its body. Any return statement that is not the last statement in a function is called an *early return*, and is an implicit task boundary (see Section 5.2). If an early return does execute, care must be taken to ensure that all tasks that implement the rest of the function body do not execute.

Return Values

As described above, tasks cannot return data values. Therefore, if a non-void function is split into tasks, some other method must be used to communicate its return value to the caller. This can be done with the addition of an extra URF register. The task containing the return statement writes the return value into this register, and the task that implements the appropriate section of the caller reads the register. Figure 6.14.
void main() {
    int a;
    a = myfunc();
    printf("%d", a);
}

int myfunc() {
    #pragma mlca Split
    return 12;
}

(a) Original C

int myfunc_Task1() {
    int retval;
    retval = 12;
    writeArg(0, retval);
}

int main_Task2() {
    int a = readArg(0);
    printf(" %d ", a);
}

(b) Resulting Sarek

int myfunc_Task1() {
    int retval;
    retval = 12;
    writeArg(0, retval);
}

int main_Task2() {
    int a = readArg(0);
    printf(" %d ", a);
}

(c) Resulting Task Bodies

Figure 6.14: Return Values
demonstrates this transformation.

Early Returns

When a function containing early returns is split into tasks, the impact of those returns on program control flow must be emulated in the resulting Sarek. Specifically, when an early return executes then all tasks derived from subsequent code must not execute. If, however, the early return does not execute then those tasks should execute.

This problem is solved by introducing a flag in the Sarek. If a task executes an early return statement it sets this flag, otherwise it doesn’t. Execution of all tasks generated from code that is after a potential early return must be made conditional based upon the value of the early return flag. Only one flag is necessary for each function that is being split, even if that function has more than one early return statement.

There are multiple methods by which this early return flag could be implemented;
these methods, as well as their pros and cons, are described in detail in Section 6.5.4. Specific examples of the handling of early returns are deferred until then.

### 6.5.3 Loop Control Statements

Sarek does not support loop `break` and `continue` statements. It is therefore necessary to emulate their behaviour if a C loop containing such statements is converted to a Sarek loop. `break` and `continue` statements can be handled much like early returns: a flag is introduced, and when a `break` or `continue` statement is executed, the flag is set. All subsequent tasks in the loop body are made conditional on that flag.

Loops can be nested, and `break` and `continue` statements only apply to the innermost loop that is currently executing. It is therefore necessary to introduce separate flags for each loop in a nest. `break` and `continue` flags, while similar, require slightly different handling. `break` statements should entirely halt execution of the loop. The loop condition evaluation task must therefore also read the flag, and incorporate the value of the flag into its calculation of the condition. In contrast, `continue` statements should only end the current iteration. The condition evaluation task should therefore not read the flag, though it should reset it. As with early return flags, specific examples are deferred until the discussion of flag implementation.

### 6.5.4 Flag Implementation

Early returns, `break` statements and `continue` statements all require the implementation of a flag that can be set by some tasks, and which will guard the execution of subsequent tasks. There are two methods by which such flags can be implemented in Sarek.

The first is to implement the flag as a control register. Tasks that need to set the flag write to the control register, and all subsequent tasks are guarded by a Sarek `if` statement. Figure 6.15 shows an example of how this strategy could be used to implement early returns. Note that the flag must be explicitly initialized before the tasks imple-
void main() {
  int a;
  myfunc(a);
  printf("%d", a);
}

int myfunc(int myarg) {
  if (myarg)
    return;
  #pragma mlca Split
  printf("%d", myarg);
  return;
}

(a) Original C

main() {
  cr_t myfunc_er_flag;
  cr_t cr1;
  reg_t a;
  myfunc_er_flag = 1;
  cr1 = myfunc_Task1(in a);
  if (cr1) {
    myfunc_er_flag = myfunc_Task2();
  }
  if (myfunc_er_flag) {
    myfunc_Task3(in a);
  }
  main_Task2(in a);
} (b) Resulting Surek

int myfunc_Task1() {
  int myarg = readArg(0);
  return (myarg != 0);
}

int myfunc_Task2() {
  return 0;
}

int myfunc_Task3() {
  int myarg = readArg(0);
  printf("%d", myarg);
}

int main_Task2() {
  int a = readArg(0);
  printf("%d", a);
} (c) Resulting Task Bodies

Figure 6.15: Early Return Handling With a Control Register Flag
menting myfunc begin execution. This is necessary for two reasons: first to provide a
default value when the early return does not execute, and second to prevent stale values
from previous executions of myfunc from interfering with the control flow. The latter is
not a potential problem in this example, but would be if myfunc were called from within
the body of a loop or if it were called multiple times.

The other potential method for implementing flags is to do so with URF data registers.
The flag can be set by writing a value to the register. All tasks that follow an early
return or loop control statement will still execute unconditionally, so their bodies must
be modified to read the flag and do nothing if it is set. Figure 6.16 demonstrates how this
method of flag implementation could be used in the presence of loop break statements.
As discussed earlier, continue statements could be handled similarly by slightly changing
the behaviour of the loop conditional evaluation task. Note that tasks after the break
statement always execute, however the flag is checked in their bodies and execution is
guarded accordingly. The guard does not surround the statements that read and write
to the URF, as those statements must execute for register renaming to proceed. In the
case that the flag is set, they will simply propagate the values in the registers unchanged.
As with the early return flag, the break flag must be reset upon exit of the loop. This
prevents stale values from being propagated to other instances of the loop.

Care must be taken when using data registers as flags to ensure that all subsequent
tasks read the flag and are guarded accordingly. If, for example, a function is called after
an early return, and that function is also split, then all of its constituent tasks must read
the caller’s early return flag. Figure 6.17 demonstrates this situation.

As shown above, flags can be successfully implemented either as data or as control
registers. The choice of which method to use is therefore not directed by the need
for program correctness; rather, it is directed by program performance. Using control
registers minimizes the number of tasks that execute – if a task is after a taken early-
return then it will not be executed at all. In contrast, if data registers are used, tasks
void main() {
    int a;

    for (a = 0; a < 10; a++) {
        if (a > 5)
            break;

#pragma mlca Split
        printf("%d", a);
    }
}

(a) Original C

main() {
    cr_t cr1, cr1;
    reg_t a;
    reg_t loop_break_flag1;

    Init(out loop_break_flag1); // zeros flag

    main_Task1(out a);
    cr1 = main_Task2(in a, in loop_break_flag1, out loop_break_flag1);
    while (cr1) {
        cr2 = main_Task3(in a);
        if (cr2) {
            main_Task4(out loop_break_flag1);
        }
        main_Task5(in a, in loop_break_flag1);
        main_Task6(in a, in loop_break_flag1, out a);
        cr1 = main_Task2(in a, in loop_break_flag1, out loop_break_flag1);
    }
}

(b) Resulting Sarek

int main_Task1() {
    int a = 0;
    writeArg(0, a);
}

int main_Task2() {
    int a = readArg(0);
    int break_flag = readArg(1);

    writeArg(0, 0); // reset break_flag
    return (a < 10 && break_flag == 0);
}

int main_Task3() {
    int a = readArg(0);
    return (a > 5);
}

int main_Task4() {
    int break_flag = 1;
    writeArg(0, break_flag);
}

int main_Task5() {
    int a = readArg(0);
    int break_flag = readArg(1);

    if (break_flag == 0) {
        printf("%d", a);
    }
}

int main_Task6() {
    int a = readArg(0);
    int break_flag = readArg(1);

    if (break_flag == 0) {
        a++;
    }
    writeArg(0, a);
}

(c) Resulting Task Bodies

Figure 6.16: Break Statement Handling With a Data Register Flag
after a taken early-return still execute and do no work. This introduces some overhead, as these tasks still read and write all of their input and output registers despite the fact that they effectively function as no-ops.

The MLCA template architecture does not specify whether or not speculative execution of tasks should be supported. If speculative execution is not supported, all Sarek control-flow statements effectively become barriers: tasks will not be dispatched across loop iteration boundaries or if/else blocks until the loop or if condition has been evaluated. In such a situation, the use of control registers as flags can hurt program performance as it introduces Sarek if statements. This can potentially add otherwise unnecessary barriers, thereby limiting the program’s parallelism.

The tradeoff between the two techniques can now be seen: using control registers minimizes task execution overhead, but can artificially limit parallelism if speculative execution is not supported. Using data registers does not limit parallelism, but does introduce added overhead for each task that does not need to execute. We have chosen to use the latter technique as current implementations of the MLCA do not support speculative task execution.
6.6 Simple Optimizations

The task splitting techniques presented in this chapter are sufficient to support the declared semantics of the split pragma, and to ensure the correctness of the resulting program. They do, however, produce sub-optimal code. Generated tasks may have unnecessary inputs and outputs, which can introduce unnecessary data dependencies. These dependencies, in turn, limit the parallelism that the program can exhibit.

In general this should not be a concern of the code generation phase. Good design practise states that it is not good to optimize too much too early. The job of optimization should mostly be left to independent downstream phases. There are a few simple optimizations, however, that fit more logically into the code generation phase. These optimizations require little or no additional analysis, and their only focus is on removing unnecessary task inputs and outputs without any additional code modifications. These optimization (some are little more than refinements of the existing techniques) are described in this section. All techniques described here have been implemented in the ATF code generator.

6.6.1 Use-Def and Def-Use Chains

*Use-Def* and *Def-Use chains* are a simple data structure for the representation of program data flow. Each statement has a set of associated nodes – one for each data location it reads (its Use nodes), and one for each data location it writes (its Def nodes). Nodes are linked together according to the data flow of the program. A Use node for a variable is linked to all Def nodes for that variable that reach it. Likewise, Def nodes are linked to all Use nodes that they reach.

Creation of Use-Def and Def-Use chains can be performed by a simple extension of the data-flow algorithm described in Section 6.3.1. Each entry in a basic block’s Gen set is a Use node, and each entry in the Kill set is a Def node. The Use nodes will be
propagated between the basic blocks through the In and Out sets, and when a Use node is killed a link is added to the Def node that killed it.

Use-Def and Def-Use chains provide a convenient representation of the program’s data flow that greatly simplifies the optimizations described below.

### 6.6.2 Unused Task Outputs

The data-flow analysis technique presented in 6.3.1 performs purely intra-task analysis. This is sufficient for determining a minimal set of task inputs – the set of upward exposed reads fully defines a task’s input requirements. The determination of task outputs is more conservative, though. It is not known what variables will be read by other tasks, so it is simply assumed that a task must output all variables that it modifies. If a variable is never read by another task (if, for example, all uses within tasks are preceded by definitions), it is still placed in the output list of the tasks that write it.

The inclusion of such unused task outputs does not hurt the parallelism of a program: since the registers are never read they cannot form true data dependencies. They can, however, cause some overhead. First, the bodies of the tasks that write them must include calls to the URF access API to write them. Second, they can increase register pressure. Many C variables are allocated to URF registers only because they are involved in unused task outputs. This can, in turn, hurt performance in a number of ways: it could cause stalls by overloading the register renaming hardware, it could necessitate a larger (and therefore less power- and cost-efficient) URF or, if that is not possible, the spilling of registers to memory.

Unused task outputs can be recognized by building Use-Def chains for the Sarek program. Use nodes are generated for each URF register that a task call reads, and Def nodes are generated for each URF register that a call writes. Once the nodes have been connected it is trivial to recognize unused task outputs. Any Def nodes that are not connected to any Use nodes can safely be removed from the program.
6.6.3 URF Copy Propagation

The use of parameter initialization tasks (see Section 6.5.1) preserves the original C program’s interprocedural data flow, but it can result in the unnecessary duplication of data values in the URF. This is demonstrated when the C code in Figure 6.18 is processed. The initial result is shown in Figures 6.19(a) and 6.19(b) after the initialization task has been merged into the first task of the callee. Note that the value stored in myfunc_p1 is a direct copy of the value stored in main_a. It is possible to eliminate that copy by simply passing main_a as a parameter to myfunc_Task2, as shown in Figures 6.19(c) and 6.19(d). In contrast, access to myfunc_p2 is left unchanged since it is not a simple copy of the value stored in main_b.

The process of finding and removing unnecessary copies, such as the one described above, is called copy propagation. It involves two main steps: the identification of tasks that create copies of URF registers, and the propagation of the source registers of those copies. The first step requires only intra-task analysis, and the second step requires only inter-task analysis.

Copy creation can be recognized by examining the Def-Use chains for each task body. Each URF read statement stores its result to a local variable, therefore it has an associated Def node. The flow of that value through the body of the task can be traced by following its Def node’s links to all corresponding Uses. Each Use is associated with a statement,
Figure 6.19: URF Copy Propagation
which itself may have zero or more associated Def nodes. These chains may simply
end within the task body, or may terminate in Use nodes associated with URF write
statements. If a chain flows from an URF read to a URF write, and all statements in
that chain create strict copies (ie. do not modify the data in any way) then a potential
URF copy has been identified. Further analysis is required, however, to verify that a
copy has indeed been created.

For copy propagation to be valid, it must be guaranteed that the definition created by
the URF read is the only definition that reaches the URF write. If other definitions reach
that node then it not guaranteed that the value that is written will always be a copy, and
propagation cannot be performed. This can be verified by following the Use-Def chains
backward, from the URF write’s Use to the URF read’s Def. If the URF read’s Def is
the only destination of all chains originating at the URF read then a true URF copy has
been created.

Once a URF copy has been identified, it is desirable to propagate the original URF
register to as many tasks’ argument lists as possible. This can be performed by generating
Def-Use chains for the Sarek program, as described in Section 6.6.2. Each task output
will then have an associated Def node. When an output is determined to be a copy, the
source of that copy can then be replaced as the input argument for all Use nodes that
are reached by the copy’s Def node.

It should be noted that URF copy propagation, as described above, can potentially
create a large number of unused task outputs. This is demonstrated by myfunc_Task1
in Figure 6.19(c). It still writes to myfunc_p1, despite the fact that myfunc_Task2 no
longer reads it. Therefore, the optimization passes should be organized such that URF
copy propagation is performed before removal of unused task outputs.
6.6.4 Using Variable Scope Information

As mentioned in Section 6.4.3, our chosen points-to analysis algorithm provides a whole-program solution. While correct, such a solution is imprecise and can report false positives. These, in turn, result in extra task input and output arguments, which can cause tasks to serialize unnecessarily. It is therefore desirable to minimize such occurrences. This can be done by refining the method in which the points-to analysis results are used.

Figure 6.20 provides an example of a situation in which straightforward use of points-to information results in unnecessary task arguments. 6.20(a) shows the C code after analysis. a and b are placed in the same alias class because they are both potential targets when f1_ptr is dereferenced. a and c are in the same alias class because they are both potential targets when f2_ptr is dereferenced. The algorithm places two variables in the same alias class by merging their respective alias classes, which effectively makes alias classes transitive. Therefore, since a and b are in the same alias class, and a and c are in the same alias class, b and c are also in the same alias class. 6.20(b) shows the C code that would result from a naive use this alias information. Note that all variables have been globalized, and the URF registers are their synchronization proxies.

It is possible to refine this result by considering the semantics of a sequential C program. The statement that dereferences f1_ptr accesses alias class 1, however it cannot legally access all members of that alias class. There is no way that it could access variable c: the statement that dereferences the pointer is in a scope that is mutually exclusive with that in which c is declared. Likewise, f2_ptr is dereferenced in a scope that is mutually exclusive with that in which b is declared.

So, when a task dereferences a pointer, it is not necessary to include all variables in the alias class that is accessed in the task's argument list. Rather, it is only necessary to include a variable from the alias class if it is:

1. Local to the function being split, OR
void main() {
    int a;
    func1(&a);
    func2(&a);
    printf("%d", a);       // AC 1
}

void func1(int *ptr_arg) {
    #pragma mlca Split
    int *f1_ptr;
    int b;
    f1_ptr = (cond) ? ptr_arg : &b; // AC 2
    printf("%d", *f1_ptr);       // AC 1
}

void func2(int *ptr_arg) {
    #pragma mlca Split
    int *f2_ptr;
    int c;
    f2_ptr = (cond) ? ptr_arg : &c; // AC 3
    printf("%d", *f2_ptr);       // AC 1
}

(a) Initial C Code

main() {
    reg_t main_a, func1_b, func2_c;
    func1_Task1(in main_a, in func1_b, in func2_c,
                out main_a, out func1_b, out func2_c);
    func2_Task1(in main_a, in func1_b, in func2_c,
                out main_a, out func1_b, out func2_c);
    main_Task3(in main_a, out main_a);
}

(b) Resulting Sarek

Figure 6.20: Imprecise Points-to Analysis
Add_Args(in: Task t, in: Statement stmt)
{
    Alias_Class ac = Get_Ac(stmt);
    Function f = Parent_Fn(t);
    foreach (Variable v in ac) {
        if (Is_Ancestral(v, f)) {
            Add_Arg(t, v);
        }
    }
}

Is_Ancestral(in: Variable v, in: Function f, out: boolean result) {
    result = false;
    Call_Graph cg = Get_Program_CG();
    if (v is in a global context) {
        result = true;
    } else if (v is local to f) {
        result = true;
    } else {
        foreach (predecessor pred to f in cg) {
            result = result || Is_Ancestral(v, pred);
        }
    }
}

Figure 6.21: Algorithm For Use of Alias Information

2. Local to a call-tree ancestor of the function being split, OR

3. Global (either C global scope, or in dynamically allocated memory).

We call non-local variables that fall into the preceding categories ancestral variables. Only local and ancestral members of an alias classes must be included as task arguments when a pointer is dereferenced. Figure 6.21 describes an algorithm for using alias information to determine task arguments implied by a given statement. It uses a simple backwards traversal of the program call graph to determine whether or not a variable is ancestral to a given statement. Note that additional care must be taken to prevent infinite recursion in the case of call-graph cycles. This situation can be handled trivially by marking visited nodes; this has not been shown in the figure for the sake of clarity.
Chapter 7

Experimental Evaluation

In this chapter we present an empirical evaluation of the performance of the ATF compiler using a variety of realistic multimedia applications. Section 7.1 presents a summary of the applications that we used. Section 7.2 contains an overview of the experimental framework that we used for compilation and simulation. Section 7.3 explains the experimental methodology, and Section 7.4 presents the performance results for each application. Section 7.5 explores the effects of the task granularity provided by the automatic task formation heuristic in further detail, and Section 7.6 evaluates the success of the MOC at processing the ATF’s output. Finally, Section 7.7 evaluates the influence of the simple optimizations presented in Section 6.6.

7.1 Applications

To evaluate the performance of the automatic task formation framework we used it to compile six multimedia applications. In this section we provide a brief overview of the purpose and structure of these applications.
7.1.1 MAD

The MPEG Audio Decoder (MAD) [5] is a program that decodes audio files in the MPEG layer 3 – MP3 – format into an uncompressed PCM output format. MP3 files contain a stream of variable-length frames. Each frame has a fixed-length header that contains, among other information, the length of the frame.

The bulk of the application is encapsulated in a loop, each iteration of which decodes a single frame. The process of decoding a frame can be divided into several phases, starting with decoding the header and ending with PCM synthesis. Header decoding is inherently serial across frames, as the location of the next frame in the input stream is not known until the length of the current frame has been read from its header. PCM synthesis is also serial across frames, as it employs an adaptive filter whose state depends upon the decoded values of previous frames. All other phases are independent across frames, however, so it is possible to begin decoding a frame as soon as the previous frame’s header has been decoded. Thus the loop can be pipelined, though care must be taken to ensure that the PCM synthesis phase is serialized across iterations.

For our experiments we decode 32 frames from a stereo input file. In the base configuration this is accomplished in 42.3 million cycles.

7.1.2 FMR

FMR [20] performs FM radio demodulation. It reads fixed-length frames from a 16-bit input stream and produces 32-bit PCM output. The application consists of a large loop, each iteration of which processes a single frame. Decoding a single frame is a largely serial process, but separate frames can be processed independently.

In our experiments we demodulate 22 frames, for a total of 50.1 million execution cycles in the base configuration.
7.1.3 GSM

The GSM application is an open-source implementation of an encoder for the European GSM 06.10 provisional standard for full-rate speech transcoding [6], intended for use in digital mobile phone networks. It compresses PCM input by translating blocks of 160 13-bit samples into 33-byte GSM frames.

The application consists of a main loop, each iteration of which generates a single GSM frame. The processing of a frame can be divided into six logical phases; a true data dependence exists across iterations for the first phase, but all future phases can be performed independently for different frames. It is thus possible to pipeline the loop. Further parallelism is available within the processing of a single frame, though it is largely contained a long-running loop with many short iterations. This loop performs a simple reduction operation on a large buffer, and has been blocked in the fully manually ported version of the application.

Our experiments use the GSM encoder to process a total of 32 frames, and the base configuration executes for 8 million cycles.

7.1.4 JPEG

JPEG is a lossy image compression algorithm, published by the Joint Photographic Experts Group [4]. The application presented here is an open-source implementation of this algorithm targeted at embedded applications [1]. It first divides its input image into 8x8 blocks of pixels called macroblocks, then compresses each macroblock.

The application consists of a main loop, each iteration of which compresses a single macroblock. This compression process has four main phases: color conversion, quantization, discrete cosine transformation (DCT) and Huffman encoding. The first three phases are independent across macroblocks. The Huffman encoding process is also logically independent across macroblocks, however the application is structured such that output
is written to a file throughout this process. Since output must be produced serially to ensure correctness the Huffman encoding phases must be serialized across macroblocks.

In our experiments we compress a 216x144 black and white image, divided into 486 8x8 macroblocks. The base configuration accomplishes this in 125.9 million cycles.

### 7.1.5 AESPKT

AESPKT is a minimal implementation of the AES (Advanced Encryption Standard) data encryption algorithm. Its input is 10 identical 16-byte AES encrypted packets. It decrypts each packet, recalculates the header, and then encrypts the packet with a new header. The application was originally written for the MLCA, and was provided by Politecnico di Milano in the form of a HASM control program and C task bodies. For our experiments we back-ported it to a pure C version; all results are presented for this back-port.

The main data structure in AESPKT is a buffer that stores the packet that is currently being processed. Decryption, modification and encryption are performed in-place in this buffer. Each packet can be processed independently, though the central buffer is reused for each one.

The base configuration of this application executes in 280,000 cycles.

### 7.1.6 MG

MG is an image-processing application that calculates the centroid and principal direction of an object. It does so by calculating the moments of order < 3 for the object. Its input is an 8x8 image made up of 1-bit pixels, and its output is a 32-bit value representing the principal direction. Similar to AESPKT, it was originally provided by Politecnico di Milano as an MLCA program. The results presented here are for a manually created C back-port.

The application works with a read-only buffer that stores the image being processed.
It calculates six moments, each of which is stored as a 32-bit integer. The principle direction is calculated from three of those moments.

The base configuration of this application executes in 6.8 million cycles.

### 7.2 Experimental Framework

#### 7.2.1 Compiler Tool-Chain

All applications were compiled with a prototype implementation of the tool-chain described in Chapter 4. The ATF component is a version of the Open Research Compiler (ORC) that has been modified to implement the task and code generation algorithms described in this document. Its input is a Sarek program and one or more C files containing the task bodies, and its output is a modified Sarek program and a single C file containing all modified task bodies. This result is passed to the MLCA Optimizing Compiler (MOC). The MOC is largely as described in [12], however some modifications were necessary to integrate it with the ATF component. The ATF component uses ORC’s whirl2c module to generate its modified C code from its intermediate representation. This code is functionally correct, however it uses non-standard idioms to represent some operations. These include the use of pointer arithmetic to represent array accesses (rather than C [] notation) and struct accesses (rather than C . and -> notation). Most modifications to the MOC were performed so that it could recognize struct accesses when they are represented as pointer arithmetic. All other modifications were minor bug fixes.

The MOC produces Sarek and C code as output. The Sarek is translated to HASM by a simple compiler that does not perform any optimization. The C code is compiled with version 3.2.2 of gcc with an ARM back-end. All C code is compiled at the -O2 optimization level.
7.2.2 The HFM Model

To simulate an MLCA system we used a functional simulator developed by ST Microelectronics, called HFM. It models a URF, control processor, one or more ARM processing units, and a simple interconnect network. It also includes both global shared memory and PU-local memory. All PUs operate on a single, global clock.

The HFM is highly parameterized – the size of the URF, number of PUs, task scheduling algorithm, cache parameters and more can all be configured at run-time through a configuration file. Table 7.1 presents the parameters that were used in these experiments.

The HFM collects a wide variety of statistics over the coarse of a simulation. We focus on two of these statistics for our measurements: the total number of global clock cycles for the entire simulation, and the number of active clock cycles for each PU. The former reflects the wall-clock time that a simulation would take if run on real hardware, and the latter reflects the utilization of each PU.

7.3 Methodology

7.3.1 Performance Measurement

Our primary metric for the evaluation of application performance is the speedup, which is the ratio of total execution cycles for a configuration compared to another configuration. A speedup greater than 1 implies that a configuration executed in fewer cycles than the base configuration, and it is therefore faster in wall-clock time. A speedup less than 1 implies that a configuration executed in more cycles than the base configuration, and is therefore slower.

Two competing factors determine the overall performance of an MLCA program. By performing parallel, out-of-order task execution the MLCA attempts to outperform a tra-
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CACHE_ENABLE</td>
<td>0</td>
<td>ARM PU cache enable/disable</td>
</tr>
<tr>
<td>DELAY_CP_FETCH</td>
<td>1</td>
<td>The number of cycles between control processor fetches</td>
</tr>
<tr>
<td>DELAY_INTERCONNECT</td>
<td>1</td>
<td>The delay for a read or write request across the interconnect network</td>
</tr>
<tr>
<td>MEMORY_LATENCY</td>
<td>1</td>
<td>The delay for accessing a single 32-bit word in memory</td>
</tr>
<tr>
<td>MEMORY_THROUGHPUT</td>
<td>1000</td>
<td>The number of words that can be accessed per cycle</td>
</tr>
<tr>
<td>NB_REG</td>
<td>5000</td>
<td>The number of URF registers</td>
</tr>
<tr>
<td>OOO_DEPTH</td>
<td>1000</td>
<td>The maximum out-of-order task execution depth</td>
</tr>
<tr>
<td>SIZE_CRF</td>
<td>16</td>
<td>The number of control registers</td>
</tr>
<tr>
<td>SIZE_CRF_KTB</td>
<td>1000</td>
<td>The size of the control register renaming table</td>
</tr>
<tr>
<td>SIZE_KTB</td>
<td>5000</td>
<td>The size of the URF register renaming table</td>
</tr>
<tr>
<td>TD_QUEUE</td>
<td>1000</td>
<td>The length of the task dispatch queue</td>
</tr>
<tr>
<td>URF_LATENCY</td>
<td>1</td>
<td>The number of cycles to access the URF</td>
</tr>
<tr>
<td>URF_NB_PORTS</td>
<td>100</td>
<td>The maximum number of URF registers that can be accessed simultaneously</td>
</tr>
<tr>
<td>URF_REG_SIZE</td>
<td>4</td>
<td>The number of bytes in each URF register</td>
</tr>
</tbody>
</table>

Table 7.1: Parameters used for the HFM in all experiments.
ditional single-processor architecture, however these improvements are offset to a certain extent by the overheads introduced by the MLCA. This includes task dispatch latency and the time tasks spend reading and writing values to and from the URF. To explore the tradeoff between parallelism and overhead we present two speedup calculations for each configuration of each application: the relative speedup and the base speedup.

The relative speedup exhibited by an MLCA program is calculated by comparing its performance on an \( N \) processor configuration to its performance on a 1 processor configuration. This provides an accurate measure of the parallelism exhibited by a program, but it masks the effects of the overhead introduced by the MLCA as it is calculated relative to a configuration that also includes those overheads.

The base speedup exhibited by an MLCA program is calculated by comparing its performance on \( N \) processors to that of a special configuration running on 1 processor. This special configuration consists of a single task that encapsulates the application’s entire functionality, analogous to a C program’s main function. Because there is only a single task in this configuration data does not need to be allocated to the URF, and the task dispatch overhead is negligible. The base speedup therefore reflects the parallelism exhibited by a specific configuration offset by the overhead of parallelization.

### 7.3.2 Application Configurations

Five of the applications presented here – MAD, FMR GSM, AESPKT and MG – were originally ported manually to the MLCA. We refer to these fully manual ports as no-pragma. The ports of MAD, FMR and GSM were produced during the development of the MOC and were used to evaluate its performance. Data for these ports is reproduced from [12]. Data for AESPKT and MG was collected specifically for this work using the ports provided by Politecnico di Milano.

We have produced a new port for each application based upon the original pure C source code. These ports consist of minimal Sarek programs that calls a single task,
which corresponds to the C `main` function. We present results for two separate configurations of this port.

In the first configuration we insert `Split` pragmas by hand and use the code generation phase of the ATF to produce a new MLCA program. The pragmas are inserted such that the tasks in this version correspond as closely as possible to those in the original, manually ported version. We refer to this configuration as `manual-pragmas`. By comparing its speedups to those of the manually ported version we can evaluate the overhead introduced by the ATF’s code generation phase.

In the second configuration we use the ATF’s automatic task formation phase to automatically insert `Split` pragmas. The task formation phase is run repeatedly until the resulting partition is deemed to be of a sufficient granularity to maximize the application’s performance. The task formation algorithm has three parameters: the splitting threshold (ST), the array score (AS) and the loop score (LS). In all cases they are set to $ST = 10$, $AS = 5$ and $LS = 5$. These values were chosen through experimentation. They require that a function contain at least two loops and/or array operations to be split. For the applications evaluated here this is sufficient to exclude trivial functions, however these values may not be suitable for all applications. We refer to this configuration as `auto-pragmas`.

The `manual-pragmas` configuration allows the evaluation of the code generation phase of the ATF framework. By comparing its performance to that of fully hand-written ports it is possible to quantify what overheads, if any, it introduces. The `auto-pragmas` configuration allows the evaluation of the effectiveness of the automatic task formation algorithm.

As discussed in Section 7.1.3, GSM contains a loop on which blocking was performed by hand in the manually ported version. The same blocking operation was performed by

---

1This is the configuration that is used to calculate all base speedups
2This iterative application of the algorithm was performed manually for this work, however it could be automated. This possibility, which is discussed in further detail in Section 9.1, is left as future work.
hand in \texttt{manual-prama} but, as it does not appear in the original application, it is not applied in \texttt{auto-prama}.

The manual ports of MAD, FMR and GSM were considered during the development of the automatic task formation heuristic; JPEG, AESPKT and MG were not. The results for the latter three applications provide insight into the general applicability of the heuristic. We avoided the creation of a manual port of JPEG, as the knowledge gained while doing so would have prevented its usefulness in this evaluation. Results are only presented for its \texttt{auto-prama} configuration.

In addition to \texttt{Split} pragmas, buffer region access pragmas \cite{12} were added manually to all configurations. This was necessitated by the fact that the MOC is unable to perform buffer region analysis on the output of the ATF. This limitation is explored in further detail in Section 7.6.

7.4 Experimental Results

Figures 7.1, 7.2 and 7.3 present the base (b) and relative (r) speedup curves for all six applications. All applications show improved performance as the number of processors is increased, which demonstrates that the MLCA is able to successfully extract parallelism from the applications. The base speedup on one processor is less than 1 for all applications except MG. This slowdown is due to the overhead introduced by the parallelization process – task dispatch delays, URF access, and extra memory allocation operations caused by buffer privatization. This overhead is minor compared to the parallelism that it enables, however, as is evidenced by the fact that all applications show a speedup of greater than 1 when they are run on two or more processors. MG shows a base speedup greater than 1 on one processor. This anomaly can be explained by the fact that its overheads are relatively small – it contains very few tasks, each task accesses only a small number of URF registers (between 3 and 6), and there is no memory allocation
Chapter 7. Experimental Evaluation

Figure 7.1: Application Speedups
Figure 7.2: Application Speedups (continued)
Chapter 7. Experimental Evaluation

Figure 7.3: Application Speedups (continued)

(a) AESPKT

(b) MG
overhead as the MOC does not privatize any buffers. Furthermore, the output from the MOC is functionally identical to the original application but it is structurally different. gcc is able to do a better job of optimizing the MOC’s output than the original source code. Similar effects are reported for some applications in [12].

For most applications the relative and base speedups exhibit similar trends, which shows that the overhead introduced by porting an application to the MLCA is a fixed value.

### 7.4.1 Performance Scalability

The performance of MAD scales well up to 4 processors, then remains largely level as more processors are added. This is due to the fact that subsequent frames can only be partially processed in parallel. PCM synthesis, which makes a significant contribution to the total processing time of a frame, must be performed serially across subsequent frames. Thus the execution of MAD can be divided into two phases. The first exhibits a large amount of parallelism as the early decoding stages of multiple frames are processed simultaneously. The second phase is fully serial, as PCM synthesis catches up and frames are finally retired. This second, serial PCM phase constitutes a critical path within the application, and therefore imposes a hard limit on the amount of parallelism that is available.

The performance of FMR scales well on 8 processors and shows no signs of flattening, which implies that it would continue to scale if the number of processors were increased. This is consistent with the structure of the application, as all input frames can be processed independently. Thus the only factors limiting the parallelism in FMR are the number of processors (and supporting hardware, such as the task dispatcher) and the number of input frames.

GSM scales well on up to 4 processors, and the manual-pragma version also shows a significant improvement between 6 and 8 processors. The initial scaling is largely due
to the parallelism available across multiple input frames. Both versions exploit that parallelism well, so they show similar scaling trends in this range. The difference in performance between the two versions on larger number of processors can be attributed to each versions’ ability to exploit parallelism within the processing of a single frame. This aspect is discussed in Section 7.5 in further detail.

AESPKT shows marked performance improvements on up to 6 processors. The speedup on 8 processors is actually slightly smaller than that on 6 processors. This can be explained by the fact that, like MAD, the application can be divided into parallel and serial phases. The serial phase occurs at application startup, as memory for several buffers is allocated. Once allocation is complete the parallel loop may begin execution, and the application enters its parallel phase. The amount of computation performed by the application is relatively small, so the overhead of the memory allocation in the initial serial phase is significant enough to form a critical path on larger numbers of processors.

MG scales well on up to 6 processors, then shows no improvement on 8 processors. This is due to the fact that the application only executes seven tasks at run-time. The first six tasks are independent, thus the program scales well up to 6 processors. The seventh task is dependent upon data produced by the first six tasks, thus it does not contribute to the parallelism of the application.

In all benchmarks the general scaling trends of the manual-pragmas and auto-pragmas versions are similar to those of the no-pragmas version. This indicates that the ATF framework is able to expose the same high-level parallelism (ie. pipelinable loops and independent tasks) that is available in the manual ports. However, in many cases the overall speedups of the versions produced by the ATF framework are lower than those of the manually-produced ports. This difference is primarily caused by two factors. First, the template-based approach used by the ATF framework to translate C semantics to Sarek results in a relatively large number of small tasks. These small tasks increase overhead by taking up scheduling resources and requiring URF access, and they contribute
little to the overall parallelism of the program. Many of these small tasks have been merged into larger tasks in the manually-ported versions, so those versions do not have the same overhead. The second factor is that the parameter initialization tasks create copies of variables, leading to duplication of data in the URF. The analyses used by the optimizer generally do not deal well with this duplication so they produce sub-optimal results. URF copy propagation mitigates this effect somewhat (see Section 7.7.2 for an evaluation of its influence), however it is not able to fully remove all duplication.

7.4.2 Evaluation of the Automatic Task Formation Algorithm

In most applications the performance of the auto-pragma version is very similar to that of the manual-pragma version. With the exception of GSM, the speedups of the auto-pragma versions on 8 processors are within one tenth of a point of those of the manual-pragma versions. This shows that the automatic task formation algorithm, despite its relative simplicity, is effective at identifying task boundaries that will expose significant amounts of parallelism to the MLCA hardware. In the case of GSM the performance of the auto-pragma is less than that of the manual-pragma version. This is due to a loop that is manually blocked in the manual-pragma version, and is not blocked in the auto-pragma version. This issue is explored in more detail in Section 7.5.

Furthermore, the automatic task formation algorithm was able to create auto-pragma versions of AESPKT and MG that exhibit good speedups at 8 processors. These applications were not considered during the development of the task formation heuristic, so this result suggests that the heuristic is not overly specific to GSM, MAD and FMR.

7.5 Granularity Experiments

The auto-pragma versions of GSM and JPEG exhibit good relative speedups but their base speedups are greatly reduced – on 8 processors GSM achieves only 1.84x and JPEG
achieves only 1.22x. It is to be expected that the base speedup for an application is smaller than the relative speedup as there is some overhead inherent in the parallelization process, but the difference between base and relative speedups is significantly larger in the auto-pragmas version of GSM than in the manual-pragmas version. A similar comparison for JPEG is not possible since there is no manual-pragmas version, however the difference between its base and relative speedups is also unreasonably large.

By examination it was found that all Split pragmas in the manual-pragmas version of GSM had corresponding pragmas in the auto-pragmas version, thus the automatic task formation was not missing any opportunities for parallelization that were exploited in the manual-pragmas version. It did, however, include some extra Split pragmas that were not featured in the manual-pragmas version. Furthermore, as discussed in Section 7.1.3, there is a loop in the original C application that is blocked in the manual-pragmas version, but not in the auto-pragmas version.

The additional Split pragmas in the auto-pragmas version of GSM suggest that the overhead that is offsetting the parallelism is introduced by the fact that the program has been partitioned into excessively fine-grained tasks. Specifically, the loop that was blocked in the manual-pragmas version was not blocked in the auto-pragmas version. The original unblocked version of this loop executes 80 iterations, and the blocked version executes 5 iterations. The body of the unblocked loop was split by the automatic task formation algorithm into 2 tasks, while the body of the blocked loop was split into a single task in the manual-pragmas version.

Both the blocked and unblocked loops are represented as C for loops. As discussed in Chapter 6, splitting a C for loop results in one or more tasks to cover the body of the loop, as well as three extra tasks – one for the for loop initialization statement, one for the evaluation of the loop condition, and one for the loop increment statement. The latter two of these tasks are executed on each loop iteration.

The blocked version of the loop therefore results in the execution of 18 tasks at run-
The unblocked version of the loop results in the execution of 322 tasks at runtime. Of those 322 tasks 162 are related to the maintenance of the loop induction variable, so their bodies consist of only one or two short statements. Relative to the amount of useful computation performed the task overhead – dispatch latency and URF access – is quite large. Their existence does not, however, limit the amount of computation that can be performed in parallel. Indeed, since they only access one URF register (the induction variable) they can be executed simultaneously with many different tasks, so they actually add slightly to the amount of computation that can be performed in parallel. This explains both the good relative speedup and modest base speedup exhibited by GSM.

To verify this explanation we produced a modified version of the GSM auto-pragma code, in which the only change was that the loop was manually blocked. The performance of this version is presented alongside that of the original manual-pragma version in Figure 7.4(a). The difference between the base and relative speedups for the new auto-pragman version is significantly reduced, and is in fact smaller than that in the manual-pragma version. It can therefore be concluded that the extra overhead observed in the original auto-pragma version was caused by overly fine-grained tasks resulting from the unblocked loop. Furthermore, on 4 and 6 processors both the base and relative speedups of the blocked auto-pragma version are better than those of the manual-pragma version. This is because the automatic task formation algorithm succeeds in finding parallelism in other sections of the application that was not exploited in the manual-pragma version (and, by extension, the fully manual port). One final item of note is that the speedup of the auto-pragma version is slightly smaller on 8 processors than on 6. This is caused by the fact that the version of the HFM used for these simulations implements a simple first-come first-served runtime task scheduler. The scheduler could be improved to eliminate this anomaly, however doing so is beyond the scope of this work.

---

31 loop initialization task, 6 condition evaluation tasks, 5 increment tasks, 5 loop body tasks, and 1 task executed after the loop to perform the final reduction operation

41 loop initialization task, 81 condition evaluation tasks, 80 increment tasks, and 160 loop body tasks
### Chapter 7. Experimental Evaluation

#### 7.1. GSM With Loop Blocking

<table>
<thead>
<tr>
<th>Number of Processors</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.00</td>
</tr>
<tr>
<td>2</td>
<td>1.00</td>
</tr>
<tr>
<td>4</td>
<td>2.00</td>
</tr>
<tr>
<td>6</td>
<td>3.00</td>
</tr>
<tr>
<td>8</td>
<td>4.00</td>
</tr>
</tbody>
</table>

- auto-pragma (b)
- auto-pragma (r)
- manual-pragma (b)
- manual-pragma (r)

#### 7.2. JPEG without DCT Splitting

<table>
<thead>
<tr>
<th>Number of Processors</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.00</td>
</tr>
<tr>
<td>2</td>
<td>1.00</td>
</tr>
<tr>
<td>4</td>
<td>2.00</td>
</tr>
<tr>
<td>6</td>
<td>3.00</td>
</tr>
<tr>
<td>8</td>
<td>4.00</td>
</tr>
</tbody>
</table>

- auto-pragma (b)
- auto-pragma (r)

---

**Figure 7.4: Granularity Results**

(a) GSM

(b) JPEG
The performance results for JPEG suggest a similar granularity problem. Its relative speedup scales well, to 3.23x on 8 processors. Its base speedup, however, only reaches 1.22x on 8 processors. A manually ported version of JPEG has not been created so it was not possible to identify the source of this discrepancy through comparison. However, by inspection it was possible to identify two sets of nested loops that account for a large number of induction variable maintenance tasks and contributed very little to the overall parallelism of the application. Each of these loop nests consists of three 8-iteration for loops, for a total of 512 iterations. This results in a total of 1026 induction variable maintenance tasks executed for each loop nest, and each loop nest is executed once for every macroblock that is processed. Since we present results for the processing of 486 macroblocks, the two loop nests result in the cumulative execution of 997,272 induction variable maintenance tasks.

We produced a new version of JPEG by simply removing the Split pragmas from the bodies of those two loop nests. The performance results for this version are presented in Figure 7.4(b). The base and relative speedups are much closer for this version, with a maximum base speedup of 3.28x on 8 processors. This figure is much more in line with what could reasonably be expected given the fact that macroblocks within a JPEG image can be processed independently.

## 7.6 MOC Integration Experiments

The MOC contains buffer region analysis capabilities that, in some situations, allow it to perform its buffer optimizations (renaming and privatization) without any programmer input. The results presented in [12] demonstrate that the success of these analyses is limited – in all cases the maximum performance provided by the MOC was achieved with at least some manually inserted directives – but the MOC was able to provide at least a modest boost in performance to MAD, FMR and GSM without any human input.
Figure 7.5: Comparison of hand-written and whirl2c-generated C code

In contrast, the MOC was completely unable to perform buffer optimizations on the output of the ATF unless all buffer region accesses were manually annotated. The task bodies in the manually ported applications consist of hand-written C code, whereas the task bodies in the ATF ported applications consist of machine-generated C produced by ORC’s whirl2c module. The machine-generated C code is functionally equivalent to the hand-written code, but it contains numerous structural differences. Of particular note is the fact that array references are generally represented in hand-written C using the \[\] operator, but they are represented by pointer arithmetic in code generated by whirl2c. This difference is demonstrated in Figure 7.5. The region analysis algorithm in the MOC is only able to recognize array references if they are represented with the \[\] operator, thus it is unable to analyze the whirl2c-generated code produced by the ATF framework.

To verify that the particular structure of whirl2c output is the only factor preventing automatic region analysis, as opposed to any functional artifacts introduced by the ATF\footnote{Such as additional task arguments, the absence of manually applied optimizations, etc.}, we produced a modified version of the ATF-generated auto-pragma version of FMR. In this version we manually changed all array references to use \[\] notation, as they are represented in the original hand-written code. All buffer region access pragmas were removed and the resulting code was used as input for the MOC. FMR was chosen because the MOC has been demonstrated to successfully perform a significant amount of automatic region analysis on the manually ported version, and because it is small enough
that all array accesses could be translated by hand in a reasonable amount of time.

The performance results for the modified version of FMR are presented in Figure 7.6. The speedup trends are very similar to those of the original auto-pragma configuration, so it can be concluded that whirl2c’s use of pointer arithmetic to represent array reference is the only factor preventing the MOC’s array section analysis. It also is of interest to note that the modified code exhibits better base and relative speedups at 8 processors than the original version – 11% better and 10% better respectively. Automatic region analysis does not find any new optimization opportunities that were not identified when manually inserted pragmas were used, so this can be explained by the fact that gcc is better able to optimize the code that uses the [] operator.

The use of pointer arithmetic to represent array indexing is not a fundamental limitation of the ATF framework. Rather, it is a limitation of the whirl2c implementation. ORC’s internal representation maintains array index information throughout all transfor-
notations. Other source-to-source compilers have shown that it is possible to automatically generate code in high-level languages such as C and Fortran that uses array indexing operators. Prime examples are the SUIF [10] and Polaris [21] compilers. Furthermore, automatic compiler passes have been developed to convert pointer arithmetic in C programs to array index notation [25]. Thus it would be possible to modify whirl2c to output array index notation, however doing so is beyond the scope of this work.

7.7 Evaluation of the Simple Optimizations

Section 6.6 presents several simple refinements to the more general code generation techniques used by the ATF framework. Here we evaluate the influence of these techniques on the overall performance of code generated by the framework.

7.7.1 Removal of Unused Task Outputs

As described in Section 6.6.2, the removal of unused task outputs does not influence the overall parallelism available in a program. Registers that are written but not subsequently read do not form true data dependences and therefore do not force the serialization of tasks. So, such task outputs have little direct influence on the performance of an application. They do however have some negative effects – they potentially increase URF register pressure and they needlessly complicate the code in the control program. It is therefore desirable to remove unnecessary task outputs when possible.

Table 7.2 quantifies the benefit of doing so. For each application it presents the number of declared URF registers both with and without the removal of unused task outputs. The transformation removes 57 - 92% of all declared URF registers for the various benchmarks. The fact that such a large number of registers can be removed is not surprising, as the base code generation algorithm adds all variables that a task writes as outputs regardless of whether or not the value is consumed. It does, however, illustrate
Table 7.2: Effects of Useless Output Removal

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>URF Regs With Removal</th>
<th>URF Regs Without Removal</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAD</td>
<td>191</td>
<td>453</td>
</tr>
<tr>
<td>FMR</td>
<td>72</td>
<td>793</td>
</tr>
<tr>
<td>GSM</td>
<td>68</td>
<td>247</td>
</tr>
<tr>
<td>JPEG</td>
<td>32</td>
<td>162</td>
</tr>
<tr>
<td>AESPKT</td>
<td>7</td>
<td>22</td>
</tr>
<tr>
<td>MG</td>
<td>5</td>
<td>62</td>
</tr>
</tbody>
</table>

the utility of this transformation.

7.7.2 URF Copy Propagation

The use of parameter initialization tasks to emulate C function-call semantics creates a
large amount of otherwise unnecessary data duplication in the URF. This duplication
influences overall application performance in two ways. Firstly (and most significantly),
it creates URF aliases for shared aggregate data structures. These aliases limit the
effectiveness of the optimizer as they force it to make conservative assumptions about
inter-task data sharing. Secondly, the duplication increases register pressure, which can
be important on instances of the MLCA with smaller numbers of registers. URF copy
propagation removes much of this duplication.

Figure 7.7 presents performance results for the auto-pragma version of GSM with
URF copy propagation both enabled and disabled. The scalability above 4 processors is
somewhat reduced, and the base and relative speedups at 8 processors are reduced by,
respectively, 19 and 18%. This is primarily due to the fact that the optimizer is unable to
perform deaggregation on several structs – it becomes confused by the URF aliases that
are created by parameter initialization tasks and conservatively skips deaggregation.
7.7.3 Using Scope to Refine Points-To Analysis

The points-to analysis algorithm used during code generation is quite efficient, however this speed comes at the cost of imprecision. It is both flow- and context-insensitive, which causes it to sometimes report that two pointers may be aliased when they are not actually. A general solution to this problem requires the use of a significantly more computationally intensive points-to analysis algorithm, however in some cases it is possible to refine the results of the algorithm by incorporating scope information from the program being analyzed.

Figure 7.8 presents the performance results for the auto-pragma version of GSM both with and without the use of scope information to refine points-to analysis. The speedup is significantly reduced: the application does not scale beyond 2 processors, and the base speedup reaches a maximum value of 0.80 at 8 processors. The use of scope information
Figure 7.8: GSM Results Without Using Scope to Refine Points-To Analysis

to refine points-to analysis is therefore critical to GSM’s performance – without it we would be better off running an unmodified version of the application on a single processor.

So, it can be seen that the lack of precision of the chosen points-to algorithm significantly limits the parallelism that the code generator can expose. Fortunately the simple refinement of using scope information to exclude some aliases increases the precision of the results to a level that enables a good degree of parallelism.

7.8 Summary

In this chapter we have explored the performance of code generated by the ATF framework. The performance of code in which Split pragmas has been inserted manually is comparable to the fully manual ports reported in [12] (and reproduced in Appendix ??),
which shows that the code generation phase introduces little overhead when compared to hand-written MLCA code. In addition, we have shown that the automatic task formation algorithm is consistently able to expose significant parallelism in the programs to which it is applied. Applications generated by the algorithm present at least as much parallelism as manually ported versions of the same application and, in the case of GSM, the algorithm actually identifies opportunities for parallelism that were overlooked in the manual port.

Heuristic algorithms, such as the automatic task formation algorithm presented here, run the risk of being tailored too closely to the specific cases that were considered during their development. In addition to the three applications that were considered during the development of the automatic task formation algorithm (MAD, FMR and GSM), we have presented results for its use on three applications that were considered only after the algorithm was complete (JPEG, AESPKT and MG). The algorithm was able to extract significant parallelism from all three applications, suggesting that it is applicable to a wide range of inputs.

We have shown that, while the automatic task formation algorithm consistently identifies the parallelism available in an application, it occasionally produces overly fine-grained tasks that introduce unnecessary overhead. Refinements to the algorithm to prevent these cases are beyond the scope of this project but would be a highly worthwhile direction for future work.
Chapter 8

Related Work

The algorithms described in this thesis, as well as the MLCA itself, have been designed specifically to exploit common characteristics of multimedia applications to provide a high level of performance with a minimal burden on the programmer. Fuhrt [17] and Fritts et al. [16] provide comprehensive overviews of the field of multimedia processing. They present design summaries for a wide range of multimedia algorithms, and highlight some of their more salient features with regards to performance. Lee et al. [24] propose the MediaBench suite as a set of benchmark applications that accurately represent the full range of operations performed by multimedia applications. Two of the MediaBench applications – GSM and JPEG – were used to evaluate the algorithms presented here.¹

A wide range of research and development has been performed in the field of automatic and compiler-aided parallelization. Much of this work has focused on scientific applications, which are characterized by significant amounts of relatively fine-grained parallelism. Small numbers of short loop iterations – on the order of tens or hundreds of cycles – are typically used as the unit of parallelization. Language extensions such as OpenMP [9] and High Performance Fortran (HPF) [2] allow a programmer to easily and concisely identify such parallel sections in their code, allowing a compiler to auto-

¹The MediaBench source code was used for GSM. An independent non-MediaBench implementation was used for JPEG, but the underlying algorithms remain the same.
matically generate synchronization and communication primatives to guarantee correct parallel execution. These language extensions reduce the tedium of writing parallel applications, but they still leave the programmer with the burden of identifying potentially parallel code sections and correctly expressing the data dependencies in that code.

Compilers such as Stanford’s SUIF [10, 26], the University of Illinois at Urbana-Champaign’s Polaris [21] and certain extensions to IBM’s ASTI [15] go one step farther and attempt to parallelize sequential applications without any programmer intervention. These compilers focus on the extensive analysis of cross-iteration data dependencies, with the ultimate goal of performing fine-grained loop transformations similar to those performed by OpenMP and HPF compilers. These compilers bear a similarity to our work in that they attempt to automatically parallelize sequential applications, however they focus on a different application space (scientific, rather than multimedia). They exploit fine- to medium-grained parallelism, while our work deals with coarse-grained task-level parallelism.

Sarkar presents a system for parallelizing programs written in the SISAL language in [28]. This approach was later adapted for use in IBM’s PTRAN [11] compiler for FORTRAN. Statements in a program are organized into a graph, all nodes of which are annotated with cost information obtained through profiling. Task partitions are then formed by iteratively grouping nodes until all nodes are contained in a single task; the partition with the smallest cost (calculated statically based upon the profiled costs) is then chosen. This approach bears some similarity to ours in that it uses compile-time cost estimation to make partitioning decisions, however its bottom-up approach to parallelization differs significantly from our top-down approach. The focus of Sarkar’s algorithm is to identify and exploit independent sections of a program’s control-flow graph. It therefore effectively identifies fully parallel sections of the program (expressable as FORTRAN DOALL loops), but does not identify partially parallel program sections (such as loops that may be pipelined). In contrast our approach is optimized for just such
Chapter 8. Related Work

partial parallelism, as it is the most common kind found in multimedia applications. Our heuristic identifies parallelism by identifying program features (such as loops and function calls) that often delineate different processing phases, rather than through CFG analysis.

The Multiscalar project at the University of Wisconsin [7] is centred upon a processor architecture that, like the MLCA, is designed to exploit coarse-grained task-level parallelism. The Multiscalar architecture provides inter-task communication through registers and through shared memory. Register communication is synchronized by hardware (possibly with compiler support), but no synchronization is performed on shared memory. Rather, Multiscalar includes an Address Resolution Buffer (ARB) to support speculative access to shared memory. Tasks perform unsynchronized access to shared memory, and if they violate a data dependence they are squashed (i.e., not allowed to commit any architectural state). The MLCA does not support any similar speculative memory access but, unlike the Multiscalar it does optimize register communication by performing register renaming.

Vijaykumar and Sohi [31] present an compiler-driven task selection algorithm for the Multiscalar architecture. Unlike the algorithms presented here they take a bottom-up approach, forming tasks by grouping basic blocks together. The speculative nature of the Multiscalar architecture requires a large amount of hardware support — tables to track control flow speculation, and buffers to store all speculative writes to memory. If these resources become full then execution must stall until space is freed up, thus Vijaykumar and Sohi’s algorithm puts an emphasis on the minimization of pressure on speculative hardware. It attempts to minimize inter-task communication (and thus maximize parallelism) by grouping the sources and sinks of data dependencies into the same tasks, however it does not make any attempt to exploit specific application characteristics to identify parallelism. In contrast, our work takes a top-down approach to parallelization. It attempts to exploit specific characteristics of multimedia applications to identify task
boundaries that will enable efficient parallel execution.

Chan and Abdelrahman’s work on the automatic paralellization of Java programs \[13\] bears a similarity to our work in that its focus is on similarly coarse-grained parallelism. Their unit of parallelism is the Java method – each method is invoked in a new thread. A static compiler performs data-flow analysis and annotates each method with a data access summary. This summary is then used by the run-time system to perform synchronization, guaranteeing correct program execution. The static analyses produce symbolic representation of a method’s data accesses, which are disambiguated by the run-time system. This prevents the need for conservative synchronization in the face of aliasing. The MLCA works very similarly, using run-time data to support efficient synchronization. It does not provide support for pointer disambiguation as Chan and Abdelrahman’s system does, but their system does not perform the functional equivalent of register renaming.

The JRPM \[14\] is a similar system for the automatic run-time parallelization of Java code. Its target is the Stanford Hydra microarchitecture \[3\], a chip multiprocessor (CMP) that supports thread-level speculation (TLS). JRPM relies on a static compiler to identify and annotate promising code sections (primarily loops) in an application. It passes an initially sequential application to the run-time interpreter, which performs value and execution frequency profiling on the annotated sections of code. Once sufficient profile data is collected the run-time system performs a cost-benefit analysis to determine what, if any, code sections are worth parallelizing. Any promising segments are recompiled to spawn speculative threads, and program execution continues. While JRPM has a similar goal to the algorithms presented here, its approach is very different. It provides no run-time synchronization (correctness is guaranteed by speculation hardware), and it performs all code transformation at run-time using profile data.
Chapter 9

Conclusion and Future Work

The MLCA is a SoC architecture intended for use primarily in embedded multimedia applications. It achieves high performance with low power consumption through the inclusion of multiple processing units. The MLCA provides a powerful programming model that eliminates much of the complexity of software development for multiprocessor systems—programmers must only express the data and control flow of their applications, and synchronization will be automatically determined and performed by the hardware at run-time.

An MLCA application consists of a control program and one or more coarse-grained computational units called tasks. Each task includes a list of registers that it reads and writes from a universal register file (URF). A control processor (CP) dispatches tasks to the various processing units according to the data and control flow specified in the control program. Tasks may be dispatched in parallel or even out of order, and the CP performs URF register renaming to eliminate any false dependencies that may exist in the original control program. As a result MLCA applications look remarkably similar to sequential applications written in traditional high-level languages such as C or Java, but they can exploit the parallelism of multiprocessor systems without modification.

Multiprocessor systems present two main difficulties for software developers: program
correctness must be guaranteed, and the application should be able to exploit the parallel hardware for high performance. The MLCA’s programming model largely eliminates the former, but careful consideration must still be given during application development to ensure that the resulting program contains tasks that can be executed in parallel. A badly chosen partition of an application might derive little or no benefit from execution on a multiprocessor system, while a different partition of the same application might show scaling performance as the number of processors is increased. Furthermore, once a program has been partitioned into tasks it must be expressed in the MLCA’s programming model. This is not a problem if the program is being developed from scratch, but the refactoring process can require significant effort for existing applications. Fortunately much of this difficulty and tedium can be eliminated with proper compiler support.

This thesis presents a set of compiler-driven techniques intended to aid software developers throughout the development process of MLCA applications. It describes an algorithm that analyzes existing sequential C code, and partitions that code into tasks such that the resulting MLCA application is likely to exhibit significant parallelism. The task boundaries produced by this algorithm are expressed as directives in the original C code, allowing developers to easily identify and, if desired, modify the results produced by the compiler. The task formation process is designed to be iterative, splitting tasks into smaller tasks until the desired level of performance is achieved.

This thesis also presents a set of code generation techniques to automatically translate annotated C code to the MLCA’s hybrid Sarek/C programming model. These techniques perform function outlining to generate task bodies, and they map C data locations (both statically and dynamically allocated memory) to URF registers. Dataflow analyses are used to determine task input and output sets, and a control program is generated to express the original program’s control and data flow. All code generation is fully automatic and requires no programmer input.

We have implemented the task formation and code generation algorithms in the Open
Research Compiler (ORC) to produce a working prototype compiler. The original input to this compiler is sequential C code, and it may be run repeatedly to perform task partitioning. The final output of this task formation phase is an annotated version of the original program, which serves as input to the code generation phase. The code generator produces a Sarek control program and a set of task bodies expressed in C. This resulting code is suitable as input for the MLCA Optimizing Compiler [12]. This prototype thus completes a compiler tool-chain that can be used to translate a sequential C application into a high performance, parallel MLCA application. This tool-chain eliminates the tedium and busy-work from MLCA application development, allowing developers to rapidly and effectively explore the design and optimization space of their application.

We have evaluated the effectiveness of both the prototype compiler and the entire tool-chain by applying them to a variety of real-world multimedia applications. The results show that the code generation phase is highly effective, and its results compare favourably with those of manually generated code when executed on an 8-processor MLCA configuration. MAD (an MPEG audio decoder) exhibited a speedup of 2.1, FMR (an FM radio demodulator) exhibited a speedup of 3.9 and GSM (voice encoding for cellular phones) exhibited a speedup of 3.1. The manually generated code for those applications showed speedups of 3.9, 4.8 and 4.0 respectively [12].

The automatic task formation phase also shows encouraging results. When applied to completely unmodified C versions of MAD, FMR and GSM it provided speedups of 2.2, 3.8 and 1.8 respectively. The speedups for the for two applications are similar to those of the manually ported versions, however GSM exhibits a lower speedup. Further experimentation showed that this discrepancy was caused by an overly fine-grained partition of a single loop, which had been manually blocked in all previously measured versions. The automatic application of loop blocking is considered orthogonal to this work, so we conclude that this granularity problem is not a shortcoming of the task formation algorithm.
The task formation algorithm is a heuristic, and therefore runs the risk of being tailored too closely to the applications considered during its development. To show its general applicability we measured its results on three previously unconsidered multimedia applications: JPEG, MG and AESPKT. After application of the tool-chain to those applications the speedups on 8 processors were 1.2, 5.9 and 3.0 respectively. JPEG suffered from granularity issues similar to those in GSM, however the problems were caused by a poor choice made by the task formation algorithm. AESPKT and MG exhibited very satisfactory performance, thus showing that the task formation has broad applicability.

9.1 Future Work

This thesis suggests several promising directions for future work. The experimental results show that both the task formation and code generation phases are able to provide competitive performance in many cases, but there are notable exceptions. Both GSM and JPEG suffer from overhead introduced by overly fine-grained tasks. Work to improve task granularity could take two complementary directions. First, while the current task formation algorithm does incorporate heuristics to limit the creation of small tasks, its approach is overly simplistic. The addition of more sophisticated techniques to estimate and control task sizes could help avoid problems such as those exhibited by JPEG. Second, it would be beneficial to implement a variety of optimizations to control task granularity. These include loop blocking, as suggested by the analysis of GSM’s performance, and could also take the form of selective task merging.

A related possibility would be the incorporation of profile data into the task formation algorithm. It is possible to run the resulting MLCA program after each iteration of task formation. If information were collected during these runs about the relative lengths of tasks it would be possible for subsequent task formation iterations to make more
accurate choices with regards to task granularity. The use of profiling data would also allow for the complete automation of the task formation process. The algorithm could be modified to run repeatedly until subsequent iterations no longer produce improvements in performance.

Another interesting direction would be to increase the cooperation between the code generation phase and the optimizing compiler. The code generation phase performs a wide variety of dataflow analyses, and much of the information that it generates could also be used by the optimizer. Sharing such information would reduce compile times by eliminating duplicate analyses, and could also provide results that the optimizer could not easily generate itself. Some information is obscured, or even lost during the translation from C to Sarek. For example Sarek does not include for loops, so the code generator must translate them to while loops. This translation makes it more difficult for the optimizer to determine some loop properties, such as trip counts. In such situations it would be possible for the code generator to perform the analysis on the original C code, and results could be passed to the optimizer via annotations in the generated code.

The experimental evaluation showed that code generation introduced little overhead in most cases, but there were notable exceptions caused by the sub-par C code generated by ORC’s whirl2c module. The structure of this code prevented the MLCA optimizing compiler from fully applying some of its optimizations, and also prevented gcc from optimizing task bodies. It would therefore be worthwhile to invest some effort to improve the quality of code generated by whirl2c.
Bibliography


