PARALLELIZATION OF MULTIMEDIA APPLICATIONS ON THE MULTI-LEVEL COMPUTING ARCHITECTURE

Utku Aydonat and Tarek S. Abdelrahman
Department of Electrical and Computer Engineering
University of Toronto
{uaydonat,tsa}@eecg.toronto.edu

ABSTRACT
The *Multi-Level Computing Architecture* (MLCA) is a novel parallel System-on-a-Chip architecture targeted for multimedia applications. It features a top level controller that automatically extracts task level parallelism using techniques similar to how instruction level parallelism is extracted by superscalar processors. This allows the MLCA to support a simple programming model that is similar to sequential programming. In order to assist programmers to easily and efficiently port multimedia applications to the MLCA programming model, a compilation environment is designed. This compilation environment enhances parallelism in MLCA programs by applying three simple code transformations that are based on known compiler optimizations. In this paper, we describe the MLCA architecture, its programming model, its compilation environment and an evaluation of its performance. Our experimental evaluation with three real multimedia applications and an MLCA simulator shows that the MLCA is a viable architecture and scaling speedups can be obtained using the compilation environment with little programmer effort.

KEY WORDS
parallel-embedded systems; compiler optimizations, privatization, parallelism enhancement.

1 Introduction

The *Multi-Level Computer Architecture* (MLCA) [1] is a novel architecture for parallel systems-on-a-chip (SOCs). It features multiple processing units and a top-level controller that automatically exploits parallelism among coarse-grain units of computation, called *tasks*, using techniques similar to those used by superscalar processors for the extraction of instruction-level parallelism. The MLCA supports a programming model that is similar to sequential programming. This model reduces programming effort, making the MLCA an attractive architecture for multimedia and streaming applications.

However, naive transformation of multimedia applications to MLCA programs may result in limited parallelism due to the use of pointers to aggregate data in shared memory. Thus, we design a compilation environment to alleviate these performance problems and to assist programmers in efficiently porting applications to the MLCA. For these purposes, three simple transformations that are based on known compiler optimizations are adopted for the MLCA programming model.

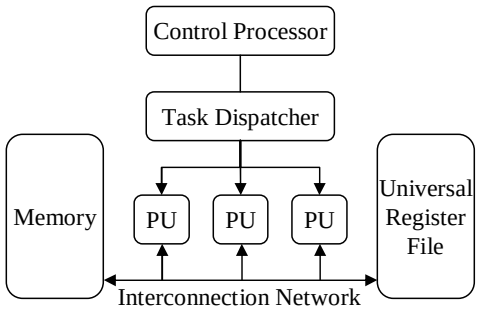
In this paper, we describe the MLCA architecture, its programming model, how naive transformation of applications to the MLCA can result in poor performance, the compiler optimizations we design to alleviate performance problems and the MLCA compilation environment used to enhance parallelism in MLCA programs. We evaluate the performance of three realistic multimedia applications using a simulator of the MLCA and a working prototype of the compiler environment. The evaluation indicates that the MLCA is an effective and promising architecture, and the compilation environment can deliver code whose performance is comparable to that of manually optimized MLCA programs.

The remainder of this paper is organized as follows. Section 2 gives an overview of the MLCA and its programming model. Section 3 motivates and illustrates the need for our compiler support. Section 4 describes our proposed compiler transformations in details. Section 5 presents our experimental evaluation. Section 6 describes related work. Finally, Section 7 gives some concluding remarks.

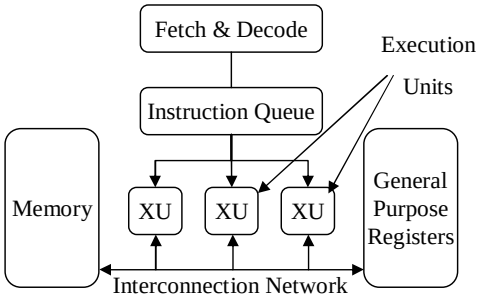
2 The MLCA

The MLCA [1] is a novel 2-level hierarchical architecture, aimed at parallel SOCs and primarily intended for multimedia applications. The lower level consists of multiple *processing units* (PUs), and the upper level of a controller that automatically exploits parallelism among coarse-grain units of computation, or *tasks*. A PU can be a full-fledged processor core, a DSP, a block of FPGA, or any other type of programmable hardware. The top-level controller consists of a *control processor* (CP), a *task dispatcher* (TD), and a *universal register file* (URF). A dedicated interconnection network links the PUs to the URF and to shared memory, as shown in Figure 1(a).

The novelty of the MLCA stems from the fact that the upper level of the hierarchy supports parallel execution of tasks, using the same techniques used in superscalar processors, such as register renaming and out-of-order execution. This leverages existing processor technology to exploit task-level parallelism across PUs, in addition to possible instruction-level parallelism within each task. The similarity of the MLCA to the microarchitecture of a superscalar



(a) MLCA.



(b) Superscalar processor.

Figure 1. Comparison between the MLCA and a superscalar processor.

processor can be seen in Figure 1.

The MLCA supports a programming model that, similar to sequential programming, does not require programmers to specify task synchronization and inter-task communication. It only requires programmers to express an application in terms of a sequential *control program*, which contains task instructions, and a set of *task functions*, each a sequential function with a specified number of input and output URF registers.

The CP executes a control program, fetching and decoding task instructions, each of which specifies a task function to be executed on a PU, as well as the inputs and outputs of the task as registers in the URF. Data dependences among task instructions are detected by identifying the source and sink registers in the URF, in the same way that dependences among instructions are detected in a superscalar processor. The CP renames URF registers as necessary to break false dependences among task instructions. Based on the dependences that occur at run-time, tasks can be issued out of order, and may also complete and commit their outputs out of order. There is also a set of control registers in the CP, which can be written to by task instructions. Logic operations (and, or) on the control registers are possible. In this paper, the task functions are assumed to be written in C, and the control program is expressed in a C-like language called *Sarek* [1].

An example of a control program is shown in Figure 2(a). It shows a single loop, executing four task instructions: `TaskA`, `TaskB`, `TaskC`, and `TaskD`. The type of access for each register is indicated as read `in` or write `out` next to the variable name that represents a URF register. `TaskD` writes to the control variable `cont` that represents a control register, and the while-loop checks this variable.

```
while(cont & 0x02)
{
  TaskA(in R1, out R2, out R3);
  TaskB(in R2, in R3, out R4);
  TaskC(in R1, out R2, out R3);
  cont = TaskD(in R3, out R4);
}
```

(a) Original code.

```
while(cont & 0x02)
{
  TaskA(in R1, out R2, out R3);
  TaskB(in R2, in R3, out R101);
  TaskC(in R1, out R102, out R103);
  cont = TaskD(in R103, out R104);
}
```

(b) After register renaming.

Figure 2. An example of control program and register renaming.

The tasks in the above example must be executed sequentially. This is because of the flow dependences between `TaskA` and `TaskB` and between `TaskC` and `TaskD`; because of the false output dependences between `TaskA` and `TaskC`, and between `TaskB` and `TaskD`; and because of the false anti dependence between `TaskB` and `TaskC`. The CP renames registers at run-time to break false dependences and thus allow some parallel execution. The control program after register renaming is shown in Figure2(b)[1]. With both false dependences eliminated, `TaskC` can be executed in parallel with `TaskA` and `TaskB`. Further, after `TaskC` writes its outputs, `TaskD` can proceed regardless of the status of `TaskA` and `TaskB`.

3 The Need for Compiler Support

The MLCA is targeted towards multimedia and streaming applications, which often apply computations to streams of data stored in memory buffers. In general, these applications tend to be modular, have relatively simple control flow, and use simple data structures, mostly C-structures and arrays [2, 3]. These structures and arrays are used as buffers to store the intermediate results of computation between consecutive stages of an application. Data exchange among these stages is accomplished by passing pointers to these buffers to functions. Such characteristics make it simple to generate an MLCA program for an application: the functions of the application are made into tasks, and function

[1]Register renaming is actually performed on the assembled machine code.

arguments are placed into URF registers to facilitate inter-task communication. Our compiler support is intended for applications already ported to the MLCA in this way, either manually by a programmer or through an automatic task generator [4].

In the MLCA programming model, tasks can communicate through the URF and/or through shared memory. URF communication is desirable for primitive data (such as integers, floats, etc.). On the other hand, shared memory communication is more desirable for aggregate data (such as buffers and structures). In this case, a pointer to data in shared memory is communicated through the URF, allowing different tasks to access the shared data.

Communication through the URF enables the MLCA hardware to eliminate false dependences by applying renaming. This is illustrated in Figure 3(a), which shows a control program with scalar URF arguments only. The renaming hardware enforces the flow dependences between `TaskA` and `TaskB`, and between `TaskC` and `TaskD`. It also breaks the false output dependence between `TaskA` and `TaskC` and the false anti-dependence between `TaskB` and `TaskC`, allowing each pair of tasks to execute in parallel.

However, communication through shared memory can result in *synchronization* and *renaming* problems, which require compiler support. The synchronization problem results from the fact that dependences among tasks caused by accesses to memory are not necessarily reflected by the dependences among pointers to memory in task arguments. This is illustrated in Figure 3(b), in which the scalar argument in Figure 3(a) are replaced by a pointer variable. Since none of the tasks write the value of `ptr` itself, `ptr` is passed as an `in` argument to all tasks. However, this causes the CP to assume that all tasks may run in parallel with one another, violating the dependences that exist.

One approach to prevent the violation of these dependences is to make `in` pointer arguments also `out` arguments, which is illustrated in Figure 3(c). Although the value of `ptr` is not modified inside any of the tasks, it is written back to the URF when each task is complete, ensuring correct execution. Regrettably, this approach results in unnecessary serialization of tasks because the `in` and `out` `ptr` argument introduces a flow dependence between `TaskB` and `TaskC`, which does not exist in the example control program of Figure 3(a) that uses scalar arguments. Indeed, making pointer arguments both `in` and `out` introduces dependences that not only cover dependences that exist in the program, but also introduces additional dependences that limit parallelism.

The renaming problem is caused by the fact that the CP only renames URF registers, but not shared memory, rendering it unable to break false dependences caused by accesses to memory. This is illustrated in Figure 3(d), which shows the earlier example control program with additional arguments used to enforce only the flow dependences that exist between `TaskA` and `TaskB` and between `TaskC` and `TaskD`. The output dependence between `TaskA` and `TaskC` and the anti dependence between `TaskB` and `TaskC` are neither detected nor resolved by the CP, resulting in incorrect execution. Additional task arguments are not helpful since the CP would rename these arguments and not the data in memory (although task arguments may be used to enforce the anti dependence, as shown in Figure 3(c), but this limits parallelism).

In conclusion, MLCA programs ported as described at the beginning of this section are likely to use pointers as task arguments, and thus will likely exhibit little or no speedup with increasing number of processors. Compiler support is needed to address this problem.

4 The Transformations

We design code transformations in order to solve the renaming and synchronization problems in the MLCA programs and, thus, to improve performance. These transformations are *parameter deaggregation*, *buffer privatization* and *buffer renaming*. They take as input a control program and the corresponding task functions and produce optimized versions of the corresponding programs.

4.1 Parameter Deaggregation

Parameter deaggregation aims to solve the renaming and synchronization problems, specifically for structures stored in shared memory. It achieves this by exposing the fields of structures in the parameter list of tasks, effectively transforming shared memory dependences into URF dependences. This enables the CP to enforce true dependences and to rename the fields of the structures, now that they are in the URF, eliminating false dependences.

Parameter deaggregation is performed by replacing pointers to structures by the fields of the structures, until all task parameters are of primitive types (e.g., `int`, `float`, `int *`, etc.). Exposing all the fields of a structure as `in` and `out` arguments of tasks can result in unnecessary dependences that limit parallelism. Therefore, we employ use-def data flow analysis [5] to ensure that a field of a structure is made an `in` argument only if it is used (before being written) by the task. Similarly, a field of a structure is made an `out` argument only if it is written to in the body of the task. This way, tasks do not have as input any field that they do not use and, similarly, they do not have any field as output that they do not define.

Three special cases are handled by parameter deaggregation. First, since structures may contain other structures or pointers to other structures in shared memory, parameter deaggregation is applied recursively. However, the transformation is not applied to recursive data structures such as linked lists and trees since the depth of these structures cannot be determined at compile-time. Second, if a structure is allocated in a task function, all the fields of the structure are made output arguments of this task, irrespective of whether they are defined in the task or not. Similarly, if a structure is

```
//Writes the value of data
TaskA(out data);

//Reads the value of data
TaskB(in data);

//Writes the value of data
TaskC(out data);

//Reads the value of data
TaskD(in data);
```

(a) URF communication.

```
//Writes the value of *ptr
TaskA(in ptr);

//Reads the value of *ptr
TaskB(in ptr);

//Writes the value of *ptr
TaskC(in ptr);

//Reads the value of *ptr
TaskD(in ptr);
```

(b) Shared memory communication.

```
//Writes the value of *ptr
TaskA(in ptr, out ptr);

//Reads the value of *ptr
TaskB(in ptr1, out ptr);

//Writes the value of *ptr
TaskC(in ptr, out ptr);

//Reads the value of *ptr
TaskD(in ptr, out ptr);
```

(c) Over synchronization.

```
//Writes the value of *ptr
TaskA(in ptr, out sync);

//Reads the value of *ptr
TaskB(in ptr, in sync);

//Writes the value of *ptr
TaskC(in ptr, out sync);

//Reads the value of *ptr
TaskD(in ptr, in sync);
```

(d) Lack of renaming.

Figure 3. Example illustrating the synchronization and renaming problems in control programs.

```
//Creates str, str->s and str->buf
TaskA(out str);

//Uses str->s->b and defines str->a
TaskB(in str, out str);

//Defines str->a and str->buf[0:19]
TaskC(in str, out str);

//Destroys str, str->s and str->buf
TaskD(in str);
```

(a) Input control program.

```
TaskA(out str, out str_a, out str_buf,
      out str_s, out str_s_b);

TaskB(in str_s_b, out str_a);

TaskC(in str_buf, out str_a, out str_buf);

TaskD(in str, in str_a, in str_buf,
      in str_s, in str_s_b);
```

(b) Output control program.

Figure 4. Parameter deaggregation example.

deallocated in a task function, all the fields of the structure are made input arguments of this task. This is to ensure that no task accesses a field of a structure before its allocation and no task accesses a field of a structure after its deallocation, and thus to ensure the proper synchronization of tasks. Third, if a section of buffer `buf`, which is a field of a structure, is defined and/or used in a task, `buf` is made both input and output arguments of the task. This is to serialize tasks that access the buffer. The unnecessary dependences caused by the possibly unnecessary `out` arguments are eliminated by buffer renaming, as will be described in Section 4.3.

Figure 4 depicts an example of parameter deaggregation for the parameter `str`, which points to a structure in memory. This structure contains an integer `a`, a pointer `buf` to a buffer, and a pointer `s` to another structure. That structure in turn contains an integer `b`. For illustration purposes, accesses by each task to the sections of buffers and to the fields of structures are indicated in the comments. Since hardware renaming is ineffective, as described in the previous section, `str` is declared both input and output to `TaskB` and to `TaskC` to ensure synchronization.

The resulting control program after parameter deaggregation is shown in Figure 4(b). In this control program, the CP can rename the `str_a` register variable (which corresponds to `a` structure field), eliminating the false dependence between `TaskB` and `TaskC`, allowing them to execute in parallel. Furthermore, `TaskB`, `TaskC` and `TaskD` will not start executing until `TaskA` completes execution, maintaining correct execution.

4.2 Buffer Privatization

Buffer privatization aims to solve the renaming problem in control programs, specifically for buffers stored in shared memory. It eliminates shared memory false dependences between collections of tasks, caused by the accesses to the same buffers, enabling the parallel execution of these tasks. This is achieved by creating additional storage for those buffers whose accesses give rise to false dependences.

Buffer privatization is performed by considering a collection of tasks $\mathtt{T}_1$, $\mathtt{T}_2$, ..., $\mathtt{T}_n$ that access a memory buffer `buf`. This collection is divided into disjoint *data access* sets: $\mathtt{S}_1$, $\mathtt{S}_2$, ..., $\mathtt{S}_k$, such that each set accesses the same data in `buf`. Thus, if a task $\mathtt{T}_i$ that uses data stored in `buf` belongs to an access set $\mathtt{S}_i$, then all tasks that possibly define the same data in `buf` also belong to $\mathtt{S}_i$[2]. A single private memory buffer (with the same size as `buf`) is created for each data access set. Accesses to `buf` in the set are replaced by accesses to this private buffer. An `Init` task is inserted before the first task in the set to create the private buffer. Similarly, a `Finish` task is inserted after the tasks complete their execution to destroy the buffer. As the result, tasks in each set access a different buffer in memory.

Buffer privatization is legal only when no flow dependences are violated. Given the definition of a data access set, flow dependences can be violated only if the tasks of a data access set are inside a loop, and loop-carried dependences exist [6]. Therefore, buffer privatization is legal when no loop carried dependences exist among tasks in the same data access set.

Buffer privatization is illustrated by the example in Figure 5.The control program in Figure 5(a) has four tasks: `TaskA`, `TaskB`, `TaskC` and `TaskD` that access a buffer `buf`. `buf` is both input and output to the tasks in order to prevent data dependence violations. Since the tasks of the control program access the same addresses in `buf`, false dependences serialize the execution of the tasks. The

[2]This is analogous to registers webs used in the context of optimizing compilers [5].

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

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

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

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

(a) Before buffer privatization.

```
//Creates a private buffer
Init(out priv1);

//Defines priv1[0:9]
TaskA(in priv1, out priv1);

//Uses priv1[0:9]
TaskB(in priv1, out priv1);

//Destroys the private buffer
Finish(in priv1);

//Creates a private buffer
Init(out priv2);

//Defines priv2[0:9]
TaskC(in priv2, out priv2);

//Uses priv2[0:9]
TaskD(in priv2, out priv2);

//Destroys the private buffer
Finish(in priv2);
```

(b) After buffer privatization.

Figure 5. An example of buffer privatization.

tasks that access `buf` can be divided into two data access sets $S_1 = \{$`TaskA, TaskB`$\}$ and $S_2 = \{$`TaskC, TaskD`$\}$ according to the data they access in `buf`. `TaskB` is in the set S_1 with `TaskA` because it uses the data defined in `TaskA`. Similarly, `TaskC` and `TaskD` form the set S_2. Buffer privatization creates a private buffer for each data access set with `Init` tasks, breaking the false dependences between `TaskB` and `TaskC` and between `TaskA` and `TaskC`. Consequently, the tasks in the two sets can execute in parallel. Since the parallelism is obtained between distinct data access sets in the body of the program, we refer to this kind of parallelism as *body-level parallelism*. The resulting control program is depicted in Figure 5(b).

Figure 6 depicts another scenario for buffer privatization. In this control program, there are three data access sets; $S_1 = \{$`TaskA, TaskB`$\}$, $S_2 = \{$`TaskC, TaskD`$\}$ and $S_3 = \{$`TaskE, TaskF`$\}$. Creating private buffers for each data access set is legal because no loop-carried dependences exists among the tasks of the sets. This exposes body-level parallelism between tasks in S_1, S_2 and S_3. Furthermore, since the `Init` task for S_2 is inserted inside the loop, a private buffer `priv2` is created in every iteration of the loop. When the CP renames `priv2` in each iteration of the loop, it will rename it along with the corresponding private buffer in the shared memory. Consequently, tasks of S_2 are assigned a private copy of `buf` in every iteration of the loop, resolving loop-carried false dependences and enabling parallel execution of task instances in different iterations of the loop.

```
//Creates a private buffer
Init(out priv1);

//Defines priv1[0:9]
TaskA(in priv1, out priv1);

//Uses priv1[0:9]
TaskB(in priv1, out priv1);

//Destroys priv1
Finish(in priv1);

while(...)
{
  //Creates a private buffer
  Init(out priv2);

  //Defines priv2[0:9]
  TaskC(in priv2, out priv2);

  //Uses priv2[0:9]
  TaskD(in priv2, out priv2);

  //Destroys priv2
  Finish(in priv2);

  //Creates a private buffer
  Init(out priv3);

  //Defines priv3[0:9]
  TaskE(in priv3, out priv3);

  //Uses priv3[0:9]
  TaskF(in priv3, out priv3);

  //Destroys priv3
  Finish(in priv3);
}
```

Figure 6. Loop-level parallelism with buffer privatization.

Buffer privatization is similar to array privatization [7], which is used by parallelizing compilers to enable the parallel execution of loop iterations. However, buffer privatization extends array privatization in that: (1) it enables the parallel execution of tasks without enclosing loops, (2) it enables the parallel execution of tasks within a loop iteration in addition to among loop iterations, and (3) it relies on the hardware register-renaming of the MLCA to allocate extra registers to hold pointers to private buffers. This is illustrated in Figure 5, in which the control program has no loops, and in Figure 6, in which parallelism exists among the iterations of the loop as well as between tasks in a single iteration.

Buffer privatization has overhead that is introduced by the additional `Init` and `Finish` tasks. Therefore, it is prudent not to apply privatization unless parallel execution can result. Figure 7 illustrates this with some examples. In Figure 7(a), tasks `TaskA` and `TaskB` form a data access set. Similarly, `TaskC` and `TaskD` are in another set. However, tasks in the two sets cannot execute in parallel because of the true dependence between `TaskB` and `TaskC` caused by the accesses to the URF scalar variable `sum`. Thus, these two data access sets are merged to form a single data access set, effectively privatizing the buffer only once inside the loop. In Figure 7(b), since the tasks of the two data access sets are accessing different regions of the same buffer, no false dependence exists among these tasks. Thus, creating separate private buffers for both data access sets is unnecessary. Finally, in Figure 7(c), the accesses to the URF variable `sum` causes a loop carried dependence between `TaskB` and `TaskA`, and, thus, prevents parallel execution. Thus, privatization is not performed for this data access set.

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

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

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

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

(a) True dependence.

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

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

  //Defines buf[10:19]
  TaskC(in buf, out buf);

  //Uses buf[10:19]
  TaskD(in buf, out buf);
}
```

(b) No false dependence.

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

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

(c) Loop-carried dependence.

Figure 7. Examples of situations in which buffer privatization is not profitable.

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

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

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

(a) Before buffer renaming.

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

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

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

(b) After buffer renaming.

Figure 8. Buffer renaming example.

4.3 Buffer Renaming

Buffer renaming addresses the synchronization problem in control programs. More specifically, this transformation renames pointer task arguments to remove dependences introduced by programmers to ensure proper synchronization of tasks, as was explained in Section 3. One example of such dependences are ones introduced by making all pointers both `in` and `out` arguments to tasks (see Figure 3(c)). We refer to a task argument which appears as an output argument only to synchronize tasks as a *synchronization output argument*, or SOA. The goal of buffer renaming is to remove dependences caused by SOAs.

Figure 8 illustrates buffer renaming with an example. The tasks `Read_Data1` and `Read_Data2` can execute in parallel, but they do not because `buf` is made an `in` and `out` argument in every task to ensure proper synchronization of the two tasks with `Write_Data`. This makes `buff` a SOA. Buffer renaming renames this SOA in task `Read_Data1` with a register variable `sync`, which is also made an `in` argument to `Write_Data`. This allows the parallel execution of `Read_Data1` and `Read_Data2` while properly synchronizing the two tasks with `Write_Data`.

Buffer renaming is performed in four steps [6]. First, array section analysis [5] is performed for each of the task functions to obtain sections of buffers accessed by each task. Second, section data flow [6] and URF scalar data flow analyses are performed on the control program to determine all pairs of tasks that can execute in parallel, and all pairs of tasks that must serialize. Third, task pairs that are serialized because of SOAs are identified. Fourth, for these tasks the SOAs are renamed to eliminate the dependences caused by them, and new synchronization dependences are created to reflect only the dependences caused by the data accesses in the program. The details of buffer renaming appear in [6].

5 Experimental Evaluation

In this section we present our experimental evaluation of the proposed transformations implemented in a prototype compiler, using three realistic multimedia applications and a simulator of the MLCA.

5.1 The MLCA Compiler

The overall structure of the MLCA Compiler prototype is shown in Figure 9. It takes as input a control program and a set of task functions. It applies the transformations described in this paper to produce an optimized control program and a set of optimized task functions. The prototype consists of two sub-compilers: a C compiler and a Sarek compiler. The C compiler analyzes the task functions, and applies inter-procedural array section analysis and inter-procedural data flow analysis on structure fields. The results of these analyses are reflected as annotations in the task functions. The annotated task functions along with the control program are given to the Sarek compiler, which uses the annotations in the task functions to apply our transformations.

A pragma-based API is also used to allow programmers to modify and/or add annotations to the task functions. These pragmas give the Sarek compiler accurate buffer section definition/use and structure fields definition/use information and can be obtained by manually inspecting the code of the application.

In our prototype implementation, the Open Research

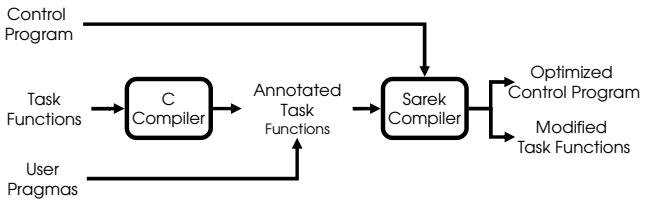


Figure 9. The structure of the MLCA Compiler prototype.

Compiler (ORC) [8] is used as the infrastructure of both the C and Sarek compilers.

5.2 Methodology

A simulator of the MLCA is used to measure and report performance. The simulator is written with C++/SystemC and reflects the overall structure of the MLCA. The interconnect among the PUs and memory adds a constant delay, and the memory model implements a simple contention mechanism, where the requests are enqueued in order and dequeued at a given rate. The simulator models the URF contention in a similar way. The simulator uses ARM processors for PUs, and tasks are compiled for ARM using the Linux-to-ARM cross-compiler 3.2.2 version of GNU's GCC using the `-O2` optimization level.

We report the performance of the applications using the *speedup*. However, since it is not possible to run a purely sequential program on the MLCA, we report the speedup of a MLCA program relative to two versions. The first is the *baseline* version of the program, in which the control program consists of a single task call to the main function of the program. This is the simplest version of an application that can run in MLCA and contains no parallelism. The speedup reported using the baseline version of the program is referred to as the *baseline* speedup. The second version is the parallel MLCA program running on a single processor, and the speedup reported with respect to this version is referred to as the *relative* speedup. Thus, baseline speedup takes into account all the factors affecting performance, such as task issue costs, and the MLCA architectural parameters. In contrast, the relative speedup reflects only the impact of the code transformations on performance.

We use three realistic multimedia applications as benchmarks to evaluate the effectiveness of our transformations: MAD, an open source MPEG audio decoder [9], FMR, an open-source program that performs FM demodulation on a 16-bit input data stream [9], and GSM, an open source implementation of the European GSM 06.10 provisional standard for full-rate speech transcoding [10]. These applications had already been manually ported and hand-optimized for the MLCA. We use these ported versions as reference, but we start from the sequential code of each application. The un-optimized versions of the applications (i.e. inputs to our compiler) are generated from the sequential code by making top-level functions of each application as tasks. Arguments of these functions are made `in` arguments and their return values are made `out` arguments. Further, global variables are transformed to `in` arguments when they are used in a function and to `out` arguments when they are defined in a function. Finally, pointer arguments are made both `in` and `out` arguments to ensure correctness, as was explained in Section 3. Tasks are then split into smaller ones by looking at the top level functions and applying the above process again. This is then repeated until the set of tasks that exist in the manually ported versions of each application is obtained, which provides us with un-optimized, but otherwise the same, versions of the applications. Given how the arguments to tasks are generated in the above process, it is natural that there is very little or no parallelism in the un-optimized versions of the applications.

5.3 Overall Performance

Figure 10 shows the baseline and relative speedups for each of the three benchmarks. The speedups are shown for versions of the benchmarks that have been manually ported and optimized for the MLCA, as well as for versions for which the code transformations have been applied by our compiler. These versions are referred to as manually-optimized (MO) and compiler-optimized (CO), respectively. In CO versions, buffer sections and structure fields access information is provided manually to the Sarek compiler through the pragmas API. Thus, the results reflect the performance of the MLCA compiler with *perfect* task analysis results.

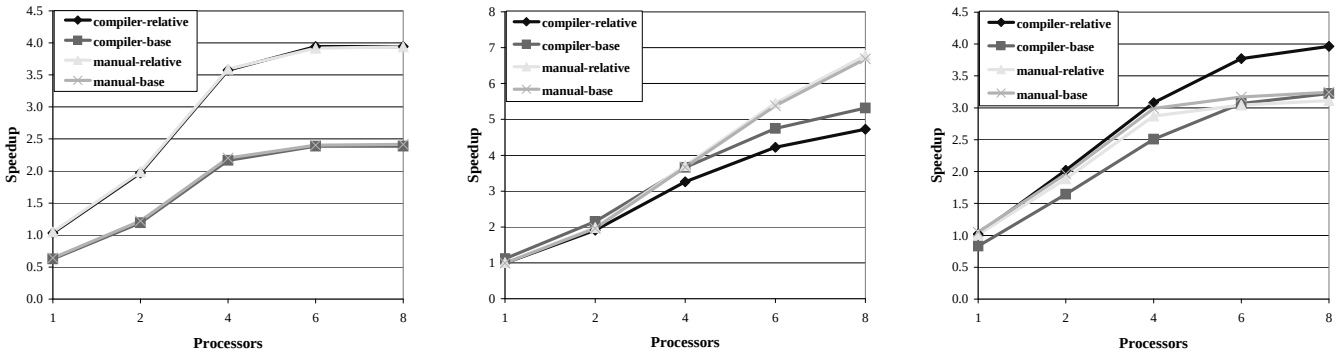
The figure shows that the manually-optimized versions of the applications exhibit scaling speedups, indicating the effectiveness of the MLCA in improving performance. Further, the figure indicates that the compiler generates code that exhibits performance similar to that of the manually-optimized versions of the applications. This indicates that the transformations are successful at improving performance.

The overheads of the transformations are also minimal; the baseline speedup of the applications is less than 1 at one processor, but not significantly, except for MAD. These overheads are further explored in Section 5.5.

5.4 Code Transformations

We experiment with four different versions of each application in order to study the impact of each of the code transformations. *OPT0* is the version in which all the code transformations are disabled; this is effectively the un-optimized version re-generated by our compiler. *OPT1* is the version to which only parameter deaggregation is applied. *OPT2* applies parameter deaggregation and buffer privatization. Finally, *OPT3* is the version to which all the code transformations are applied, and, thus, is the same as the compiler-optimized version. Figure 11 shows the execution cycles on 8 processors of each version, normalized with respect to the OPT0 version.

The figure shows that parameter deaggregation has no impact on the execution cycles of FMR, because FMR does not use any structures. On the other hand, it speeds up



(a) MAD. (b) FMR. (c) GSM.

Figure 10. The baseline and relative speedups of each of the benchmark programs, for both manual and automatic application of the transformations.

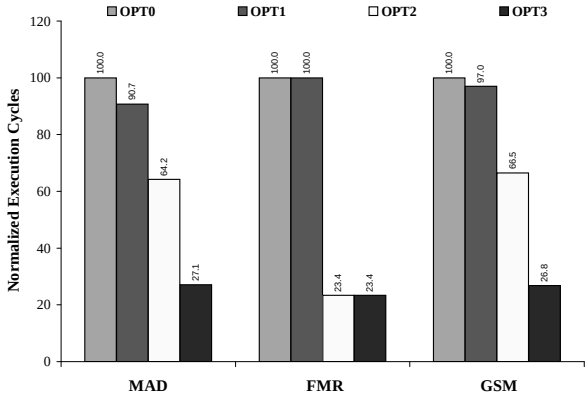


Figure 11. The effect of the code transformations on the total execution cycles.

MAD by 9.3% and GSM by 3%. Further, deaggregation opens up more opportunities for the buffer transformations, by extracting buffers inside structures. Buffer privatization reduces the execution cycles by 26.5% in MAD, 76.6% in FMR and 30.5% in GSM after parameter deaggregation is applied. These improvements are due to 18 buffers privatized in MAD, 21 buffers privatized in FMR and 15 buffers privatized in GSM. Buffer renaming improves the performance of MAD by 37.1% and the performance of GSM by 39.7%. These improvements are due to a large number of tasks that access different sections of a buffer. Thus, all three transformations are effective in improving performance when applicable.

5.5 Transformation Overheads

In order to evaluate the overhead of our transformations, we measure the difference between the total execution cycles of `OPT0` and `OPT3` versions obtained with one processor. We find that, normalized with respect to their baseline versions, 10.6%, 0.5% and 2.7% overheads are introduced by our transformations to MAD, FMR and GSM applications respectively. The causes of these overheads are the additional `Init` and `Finish` task calls introduced by buffer privatization and the increased number of task arguments introduced by parameter deaggregation and buffer renaming.

5.6 The ORC Compiler

We enhanced the ORC infrastructure to perform array section analysis for pointer function arguments and data flow analysis for structure fields [6]. In order to assess the ability of this enhanced infrastructure in performing the analyses necessary for our transformations, we generate three versions of each application. The *PRAGMA1* is a version in which all buffer sections and structure fields definitions/uses are generated by ORC. The *PRAGMA2* is a version in which only buffer sections for buffers defined inside structures are provided manually. In addition, since the ORC array section analysis is flow-insensitive, manual pragmas are inserted to prevent buffer sections to be conservatively marked as used, when sections of a buffer are only defined. Finally, the *PRAGMA3* is a version in which all the buffer section and structure fields definition/use pragmas are provided manually. This version is the same as the CO version described earlier.

Figure 12 depicts the execution cycles on 8 processor for each version compared to the un-optimized version (OPT0). The figure shows that for MAD and GSM applications, speedups of 9.3% and 3.0% are respectively obtained with the PRAGMA1 versions. These speedups are due to the accurate structure fields data-flow analysis performed by ORC, enabling only parameter deaggregation. Further, the PRAGMA1 version of FMR exhibits 43.5% of performance improvement compared to the un-optimized version. This shows that ORC is able to accurately generate some of the buffer sections required for the transformations. However, there also exist sections that are estimated conservatively, preventing buffer privatization and buffer renaming from being applied in some cases.

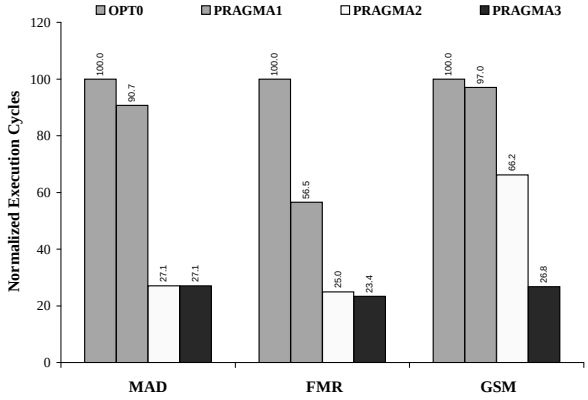


Figure 12. The effect of compiler pragmas on the total execution cycles.

On the other hand, since the majority of the buffers are defined within structures and excessive pointer arithmetic is used to access the buffers of the MAD and GSM applications, the array section analysis of the ORC is unable to provide the MLCA Compiler with accurate buffer sections. Consequently, buffer privatization and buffer renaming could not be applied. However, when the buffer sections are provided manually for the buffers in structures and for flow-insensitive sections performance improves significantly for MAD and GSM, as indicated by the PRAGMA2 version. The additional improvements of the PRAGMA3 version for GSM is caused by a large number of buffers for which the sections are only predicted conservatively by the ORC. This is due to, again, the excessive pointer arithmetic usage in GSM.

These results show that ORC is generally able to provide the analyses needed for our transformations, with two exceptions: sections of buffers defined within structures and flow-sensitive section usage. We believe these exceptions can be easily handled within the framework of ORC to enable the full automation of the transformations.

6 Related Work

Our transformations build on and extend compiler transformations proposed in the literature for extracting loop-level parallelism and for locality enhancement [7, 11, 12]. Specifically, our transformations extract parallelism at coarser grain (tasks vs. loop iterations) and they require no loops. When loops are present, our transformations can extract parallelism both within and across loop iterations, thus extending previous work.

Chilimbi et al. [11] proposes structure splitting, which is analogous to our parameter deaggregation transformation, for the purpose of increasing cache locality. They identify infrequently accessed structure/object fields and extract/move them from the structure/object. In contrast, we segregate all structure fields into scalars to enhance parallelism, and also apply the transformation recursively to maximize parallelism.

Array privatization [7, 13] is an optimization technique to improve loop-level parallelism in programs, and is employed in many parallelizing compilers [14, 15] and programming interfaces [16]. Our buffer privatization bears similarities to array privatization, but differs in the granularity of parallelism. Array privatization privatizes arrays for entire loop iterations, while buffer privatization privatizes buffers for sets of tasks, irrespective of whether they are enclosed by a loop or not, and not necessarily for an entire loop iteration. This enables buffer privatization to extract more parallelism, since each buffer may be privatized several times in a single loop iteration, resulting in parallelism within a single iteration of a loop. The differences make buffer privatization unique in its design and application to MLCA programs.Furthermore, buffer privatization relies on the MLCA renaming hardware to simplify code generation [6].

Our buffer renaming transformation aims to remove unnecessary synchronization arguments, thus increasing parallelism. There has been considerable work on removing unnecessary synchronization in parallel programs, for example for removing synchronization in Java programs [17, 18, 19]. These works aim to remove synchronization primitives that do not contribute to the correct execution of a program (e.g. nested synchronization). In contrast, our work detects superfluous variables (i.e. SOAs) used by programmers to synchronize parallel tasks and removes them when they limit potential parallelism.

In our annotations, we use representations of array sections similar to *simple sections* proposed by Balasundaram and Kennedy [20]. Further, we employ an interprocedural section dataflow analysis to determine buffer sections used by tasks. This analysis is done by ORC and is similar to those given in [21, 22, 23]. We use these sections to determine dependences among tasks using similar algorithms to the ones used by Li et al. [24, 25].

7 Concluding Remarks

In this paper, we described the compilation environment designed to improve the performance of MLCA programs. The compilation environment enhances parallelism in MLCA programs by applying three simple code transformations.These transformations are based on known compiler analyses and eliminate the synchronization and renaming problems in MLCA programs caused by the use of pointers to shared memory in task arguments. An API was also provided by the compilation environment to allow programmers to provide high-level data usage information, which can reasonably be obtained from an application's code.

We evaluated the MLCA and its compilation environment using three real multimedia applications and a simulator of the MLCA. Our evaluation indicates that scaling performance can be obtained when applications are hand-ported and optimized. Further, the performance of the MLCA programs ported using the compilation environment is comparable to that of the hand-optimized applications,

when array access information is available for the tasks. Finally, inter-procedural array section analysis applied by the ORC is too conservative and does not produce perfect buffer sections in some applications. Thus, programmer effort in porting applications to the MLCA can be further reduced when this analysis in ORC is improved.

Future work includes improving the precision of array section analysis in ORC, evaluation with more industrial-strength multimedia applications, and integration with the automatic task generation phase already in progress [4].

References

[1] F. Karim, A. Mellan, A. Nguyen, U. Aydonat, and T. Abdelrahman, "A multi-level computing architecture for multimedia applications," *IEEE Micro*, vol. 24, no. 3, pp. 55–66, 2004.

[2] C. Lee, M. Potkonjak, and H. Mangione-Smith, "Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems," in *Proc. of the Int'l Symposium on Microarchitecture*, pp. 330–335, 1997.

[3] C. E. Kozyrakis and D. A. Patterson, "A new direction for computer architecture research," *IEEE Computer*, vol. 31, no. 11, pp. 24–32, 1998.

[4] K. Stewart, "Automatic task formation techniques for the MLCA," Master's thesis, Department of Electrical and Computer Engineering, University of Toronto, 2006.

[5] S. Muchnick, *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[6] U. Aydonat, "Compiler support for a multimedia system-on-chip architecture," Master's thesis, Department of Electrical and Computer Engineering, University of Toronto, 2005.

[7] P. Tu, *Automatic Array Privatization and Demand Driven Symbolic Analysis*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1995.

[8] The ORC Compiler. `http://ipf-orc.sourceforge.net`.

[9] A. Mellan, *Personal communication*. 2003.

[10] The MediaBench. `http://cares.icsl.ucla.edu/MediaBench/applications.html`.

[11] T. M. Chilimbi, B. Davidson, and J. R. Larus, "Cache-conscious structure definition," in *Proc. of the ACM SIGPLAN Conf. on Prog. Lang. Design and Implementation*, pp. 13–24, 1999.

[12] R. Eigenmann, J. Hoeflinger, Z. Li, and D. Padua, "Experience in the automatic parallelization of four Perfect-benchmark programs," in *Proc. of the Workshop on Prog. Lang. and Compilers for Parallel Computing*, pp. 65–83, 1991.

[13] Z. Li, "Array privatization for parallel execution of loops," in *Proc. of the ACM Int'l Conf. on Supercomputing*, pp. 313–322, ACM Press, 1992.

[14] T. P. C. Forum, "PCF parallel Fortran extensions," *SIGPLAN Fortran Forum*, vol. 10, no. 3, pp. 1–57, 1991.

[15] L. J. Toomey, E. C. Plachy, R. G. Scarborough, R. J. Sahulka, and J. F. Shaw, "IBM parallel Fortran," *IBM Systems Journal*, vol. 27, no. 4, pp. 416–435, 1988.

[16] The OpenMP. `http://www.openmp.org`.

[17] J. Bogda and U. Hözle, "Removing unnecessary synchronization in Java," in *Proc. of the ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 35–46, ACM Press, 1999.

[18] M. Stoodley and V. Sundaresan, "Automatically reducing repetitive synchronization with a just-in-time compiler for Java," in *Proc. of the Int'l Symposium on Code Generation and Optimization*, pp. 27–36, IEEE Computer Society, 2005.

[19] E. Ruf, "Effective synchronization removal for Java," in *Proc. of the ACM SIGPLAN Conf. on Prog. Lang. Design and Implementation*, pp. 208–218, ACM Press, 2000.

[20] V. Balasundaram and K. Kennedy, "A technique for summarizing data access and its use in parallelism enhancing transformations," in *Proc. of the ACM SIGPLAN Conf. on Prog. Lang. Design and Implementation*, pp. 41–53, 1989.

[21] R. Rugina and M. Rinard, "Symbolic bounds analysis of pointers, array indices, and accessed memory regions," in *Proc. of the ACM SIGPLAN Conf. on Prog. language design and implementation*, pp. 182–195, ACM Press, 2000.

[22] B. Creusillet and F. Irigoin, "Interprocedural array region analyses," in *Int'l Workshop on Languages and Compilers for Parallel Computing*, pp. 4–1 to 4–15, August 1995.

[23] Y. Paek, J. Hoeflinger, and D. Padua, "Simplification of array access patterns for compiler optimizations," in *Proc. of the ACM SIGPLAN Conf. on Prog. lang. design and implementation*, pp. 60–71, ACM Press, 1998.

[24] J. Gu and Z. Li, "Efficient interprocedural array dataflow analysis for automatic program parallelization," *IEEE Trans. Softw. Eng.*, vol. 26, no. 3, pp. 244–261, 2000.

[25] J. Gu, Z. Li, and G. Lee, "Symbolic array dataflow analysis for array privatization and program parallelization," in *Proc. of ACM/IEEE Conf. on Supercomputing*, p. 47, December 1995.