Abstract

Single thread performance improvement using more complex structures and higher frequencies is currently reaching its limits. As such, several architectures that target this problem through exploiting coarse grained multithreading have been proposed. Such an architecture is the Data-Driven Multithreading Chip Multiprocessor (DDMCMP) which is based on a dataflow-like model of execution.

To fully explore the benefits of a new architecture an automated compiler is required. As a fully automated tool requires tremendous effort, semi-automated solutions are often the first approach. In this paper we present the DDM C Pre-Processor (DDMCPP), a specialized tool that allows easy programming for the DDM-CMP architecture. The only responsibility of the programmer is to express the available parallelism using special directives. DDMCPP then converts the code into a C program able to compile for the DDM-CMP architecture using a commodity compiler.

1. Introduction

Architecture and technology advances have resulted in microprocessors that are able to achieve very high performance. Nevertheless, exploiting this performance for real-world applications is an on-going challenge. A common case for enhancing performance is by overlapping different tasks, i.e. executing them in parallel. Substantial effort has been put into developing hardware techniques [3] as well as compilers [10] that transparently exploit parallelism at different levels (e.g. instruction and task). This type of parallelism that is exploited transparently, i.e. without user intervention, is known as implicit parallelism.

Well known hardware techniques that successfully exploit parallelism in a transparent way include pipelining, out-of-order execution, superscalar execution, and the Tomasulo Algorithm which allows for the dynamic instruction reordering [8].

In terms of the software techniques that exploit implicit parallelism, there is extensive on-going effort in the development of parallelizing compilers such as Polaris [1], as well as compilers that schedule multiple instructions in parallel for VLIW architectures such as Impact [2].

While implicit parallelism is the desired technique to exploit performance of real-world applications, currently it is limited to a moderate level of parallelism. Therefore, in order to achieve high degrees of parallelism, the user intervention is required leading to what is known as explicit parallelism. With this approach the user identifies the parallel sections of the code and inserts commands that allow the system to execute the tasks in parallel.

Notice that the explicit parallelism approach does not exclude the use of implicit parallelism. Effectively, the user is able to determine the tasks that are able to execute in parallel and leave the system to further exploit parallelism at a lower level (e.g. instruction-level parallelism).

It is possible to express explicit parallelism using one of the two well known models: shared memory or message passing. For the former it is possible to use, among others, Unix processes or threads (e.g. pthreads). As for the latter, it is possible to use, for example, Unix sockets. In order to help the programming task, higher-level APIs and/or libraries have been developed for both models being the most popular OpenMP for the shared-memory and MPI for the message-passing model.

While it is already difficult to express parallelism using the above mentioned libraries on existing systems, this task is considerably more challenging when trying to exploit parallelism for newly proposed architectures. These architectures are in most cases implemented using simulators and compilers do not usually exist for them. Therefore, in order to evaluate a new architecture, researchers are usually forced to hand-code the applications in order to efficiently exploit the benefits of their proposal. This is usually a very time consuming process that becomes impractical and unfeasible when evaluation is done using a large number of applications. The alternative option, pursued by
some projects, is to develop a compiler for the particular architecture that is able to automatically generate the parallel code from the existing serial applications. Although being the ideal solution, this is a very time consuming effort that may take precious time, delaying the development of the architecture.

In this work, we present our approach to avoid the above mentioned problem. Within our group we proposed the Data-Driven Multithreading programming model (DDM) [5], the Data-Driven Network-of-Workstations (D²NOW) [4] and Data-Driven Multithreading Chip-Multiprocessor (DDM-CMP) [6]. The first evaluations of the model using the D²NOW architecture were done using applications that were hand modified. For DDM-CMP we opted to build a compiler. Nevertheless, in order to reduce the development time, we decided to start with a Pre-Processor that takes as input regular C code augmented with special directives and produces the parallel code for the target architecture. We call this tool the Data-Driven Multithreading C Pre-Processor or DDMCPP. The advantage of DDMCPP is that the user may express the parallelism using a set of simple directives, therefore accelerating the process of parallelizing the applications for the DDM-CMP architecture. In addition, while using DDMCPP it is possible to better understand the tasks that a fully automated compiler should be responsible for. As such, we may use DDMCPP as a tool to help the development of the compiler as well.

Using DDMCPP allows the user to focus on expressing the parallelism. All other details, like sending and receiving data or synchronizations, are handled automatically either by the pre-processor or by the model. Although the same results could be achieved by using existing models such as OpenMP or MPI, DDMCPP explores the characteristics of the DDM model in order to offer a more intuitive way to design parallel applications.

The fact that in DDM data dependencies are inherent in the model allows DDM binaries to be more efficient as they do not need handle synchronization events. We believe that this characteristic of DDM allows the programmer to express parallelism in a more intuitive way compared to other similar schemes like OpenMP or MPI. Such a comparison however, is subject of future work and will not be covered in this paper.

This paper is organized as follows: Section 2 gives a brief overview of the DDM model of execution, the DDM-CMP architecture and its runtime system. Section 3 describes the Pre-Processor directives and Section 4 its operation. Section 5 goes over the process of transforming an application to be executed on the DDM-CMP architecture using a code example. Section 6 discusses the future directions for the tool and finally Section 7 presents the conclusions for this work.

2. Data-Driven Multithreading

2.1. DDM Model of Execution

The Data-Driven Multithreading (DDM) model of execution has been evolved from the dataflow model of computation. DDM provides effective latency tolerance by allowing the computation processor produce useful work, while a long latency event is in progress. This is achieved by scheduling a thread for execution only when its input data have been produced i.e. scheduling in a Data-Driven manner.

A program in DDM is a collection of Code-Blocks. Each Code-Block comprises of several threads where a thread is a sequence of instructions of arbitrary length. A producer/consumer relationship exists among threads. In a typical program, a set of threads, called the producers, create data used by other threads, called the consumers. Scheduling of Code-Blocks, as well as scheduling of threads within a Code-Block is done dynamically at runtime according to data availability. This task is performed with the help of the Thread Synchronization Unit (TSU) as described in the next section. The instructions within a thread are fetched by the CPU sequentially in control-flow order. Nevertheless, the CPU can reorder the sequence of instructions internally to exploit the advantages of out-of-order execution.

2.2. Hardware Support for Data-Driven Multithreading: Thread Synchronization Unit

The synchronization information, i.e. the information regarding the data dependences among the threads, is determined at compile time. The dependencies for a certain thread are expressed by the producer and the consumer threads. This event is used by the compiler to create the thread templates. The thread template include the thread unique identifier (threadID), the number of producer threads (ready count) and the list of consumer threadIDs (consumer list).

The Thread Synchronization Unit (TSU) is the hardware unit responsible for the scheduling of the DDM threads [7]. The CPU communicates with the TSU through simple read and write instructions as the TSU is implemented as a memory mapped device. As such, it is possible to use commodity CPUs.

The sequence by which the processors execute the program threads is defined dynamically by the TSU according to data availability, a thread can be executed only when all its producers have completed their execution. When a thread completes its execution it notifies the TSU. This event together with the program synchronization information allows the TSU to dynamically identify the next threads.
to be executed.

2.3. The Data-Driven Multithreading Chip Multiprocessor

DDM-CMP is a chip multiprocessor able to support the Data-Driven Multithreading model of execution. DDM-CMP combines the benefits of the DDM model together with those of the CMP architecture without requiring any modification to the Operating System or the CPUs. Its performance and power reduction potential have been presented in our previous work [6, 9]. The main factor that allows the DDM-CMP architecture to achieve high performance benefits is that it utilizes the DDM model of execution, which allows it to explore more parallelism. Using simple cores is the main reason that enables the architecture to achieve power reductions.

The DDM-CMP chip includes the execution cores along with all other units that are required in order to operate as a shared-memory chip multiprocessor and support the DDM model of execution. Without loss of generality, Figure 1 depicts the layout of a DDM-CMP chip with only 4 cores.

![Figure 1. The layout of DDM-CMP chip with 4 cores.](image)

2.4 DDM-CMP Runtime System and Support

Although it is not a priority of this work to cover the details of DDM-CMP, it is important to describe the Runtime Support System. This is because its code must be embedded into the application, and this is one of the tasks of the pre-processor.

A primary target of the DDM-CMP architecture is to be able to execute not only DDM applications, but also conventional, non-DDM binaries. To meet this goal, a Runtime Support System (RSS) that does not require modifications of the Operating System or the CPU cores has been designed [7]. As such, the RSS has to satisfy two important requirements. First, when an application is executed in parallel in a shared-memory multiprocessor, the execution CPUs need to have access to the same virtual address space. This is also true for the DDM-CMP architecture. Secondly, as the TSU space is limited, a mechanism that dynamically loads and unloads its internal structures with the proper data is required.

To meet these requirements, we designed a simple, lightweight user level process, the DDM-Kernel [7], which is presented in the next section.

2.4.1 The DDM-Kernel

A DDM application starts its execution by launching $n$ DDM-Kernels. Each Kernel is executed by a different process on a different CPU. The application completes its execution when all its kernels have done so. This approach guarantees a common virtual address for all CPUs, the first requirement the RSS must meet.

Figure 2 depicts the pseudocode of the DDM-Kernel. Its first operation is to transfer the execution to the address of the first instruction of the Inlet Thread (the first thread of each Code-Block is called the “Inlet Thread”) of the first Code-Block it will execute. The rest of its code is a simple loop, which purpose is explained later on.

The primary responsibility of Inlet Threads is to load the TSU with all threads of their Code-Block (Figure 3-(a)). The information regarding these threads is inserted in the application’s code during compilation. At this point, the operation of the DDM-Kernel guarantees that the TSU will contain the necessary information to execute all threads of that Code-Block.

On the other hand, the last thread of a Code-Block is the block’s Outlet Thread (Figure 3-(b)). Its primary operation is to clear the resources allocated on the TSU for that block. When such a command is sent to the TSU, all its internal state is flushed. The inlet and outlet threads enable the DDM-Kernel to meet its second goal, the dynamic loading/unloading of the TSU.

```python
goto (firstInstructionINLET_THREAD)
```

```plaintext
THREAD_SELECT:
    address = readReadyThreadFromTSU();
goto address;
```

Figure 2. The pseudocode of the DDM-Kernel.

The THREAD_SELECT loop, combined with the operation of the TSU, guarantee that execution will be transfered to subsequent threads. Specifically, the last operation of all threads is to notify their TSU that they have completed their execution and jump to a special loop in the DDM-Kernel.
named the \textit{THREAD\_SELECT} loop (Figure 3-(a)-(b)-(d)). Acknowledging the thread completion is achieved by sending a special flag to the corresponding TSU. This triggers the TSU operation that identifies the next ready thread.

\begin{verbatim}
for all threads \text{t} of my code_block {
    loadTSU(\text{t});
    exec_Completed();
    goto THREAD\_SELECT;
}
\end{verbatim}

\begin{verbatim}
clearTSU();
exec_Completed();
go to THREAD\_SELECT;
\end{verbatim}

\begin{verbatim}
instruction 1;
instruction 2;
. . .
instruction \text{n};
exec_Completed();
go to THREAD\_SELECT;
\end{verbatim}

\begin{verbatim}
exit;
\end{verbatim}

\begin{figure}[h]
\centering
\begin{tabular}{|c|c|}
\hline
\textbf{Inlet Thread} & \textbf{Outlet Thread} \\
\hline
for all threads \text{t} of my code_block {
    loadTSU(\text{t});
    exec_Completed();
    goto THREAD\_SELECT;
}&
\begin{verbatim}
clearTSU();
exec_Completed();
go to THREAD\_SELECT;
\end{verbatim} \\
\hline
\end{tabular}
\caption{The pseudocode of DDM threads. The first thread of a block is named \textit{Inlet Thread} and the last \textit{Outlet Thread}.}
\end{figure}

The \textit{THREAD\_SELECT} loop performs two simple operations: reads from the TSU the address of the next available thread and branches to that address. When the CPU requests this address, the TSU first tests whether a ready thread exists. If this is the case, it returns the address of its first instruction. In case no ready thread exists, it returns the address of the \textit{THREAD\_SELECT} loop (Figure 3-(d)).

As it was mentioned earlier, upon their completion, all threads jump to the address of the \textit{THREAD\_SELECT} loop. However, the outlet thread of the last block, is set statically to force the DDM-Kernel to exit (Figure 3-(c)).

3. Data-Driven Multithreading C Directives

A primary target of the DDM Pre-Processor is to hide the details of the Runtime Support System from the programmer. To develop a parallel application the programmer only needs to describe the parallel sections of the program and needs not worry about runtime tasks, such as loading and unloading the TSU.

3.1. DDM Code-Block

As explained earlier all DDM-Threads must be defined inside a DDM-Block. As such, directives to define the boundaries and structure of such blocks are required.

\begin{verbatim}
#pragma ddm block B_ID
[import (T1 A1, Tn An)]
[export (B1:T_ID1, Bn:T_IDn)]
\end{verbatim}

Defines the start of the DDM Code-Block \textit{BJD}, where \textit{BJD} may have any value from 1 to MAX\_BLOCKS. Optionally the programmer may define variables to be imported to the Code-Block, \textit{i.e.} make them accessible to all threads of the block. We represent these with the \textit{import} list \textit{Ti Ai} where \textit{Ti} represents the type of the variable (\textit{e.g.} int or float) and \textit{Ai} represents the name of the variable. Also, the programmer may define variables to be exported from the Code-Block. These are represented by the export list \textit{Bi:T_IDi} where \textit{Bi} represents the name of the variable and \textit{T_IDi} the thread that has produced this variable.

\begin{verbatim}
#pragma ddm endblock
\end{verbatim}

Defines the end of the code of the DDM Code-Block whose code started with the previous \texttt{pragma ddm block} directive.

3.2. DDM Thread

\begin{verbatim}
#pragma ddm thread T_ID kernel K_ID
[import (B1:T_ID1, Bn:T_IDn)]
[export (T1 A1, Tn An)]
\end{verbatim}

Defines the DDM Thread \textit{TJD}. This value must be between 1 and MAX\_THREADS. This Thread will be executed by DDM-Kernel \textit{KJD}. This value must be between 1 and KERNELS where KERNELS is a command line option of the pre-processor and shows the number of DDM Kernels.

Optionally the programmer may define a list of variables that are imported from other threads. These are represented by the import list where each element is of the form \textit{Bi:T_IDi}, \textit{Bi} representing the name of the variable and \textit{T_IDi} the identifier of the thread that has produced the \textit{Bi} variable. Note that if a thread needs to consume a variable that has been imported into the block with the import list of the block directive, the programmer needs to also specify that variable in the thread import list and the \textit{TJD} for that variable should be zero.

Also, the programmer may specify a list of variables that should be exported from the thread, \textit{i.e.} the variables produced by this thread and consumed by some other thread. In this list each element is represented by \textit{Ti Ai} where \textit{Ti} represents the type of the variable (\textit{e.g.} int or float) and \textit{Ai} the name of the variable.

\begin{verbatim}
#pragma ddm endthread
\end{verbatim}

Defines the end of the code of a DDM Thread whose code started with the previous \texttt{pragma ddm thread} directive.
3.3. DDM Loop

```c
#pragma ddm for thread ( t1, ... tn )
   kernel ( k1, ... kn )
   index <var>
   <num1>
   <num2>

#pragma ddm endfor
```

Defines the loop body code which will be executed by threads T1 and Tn on kernels K1 to Kn. The loop index variable var ranges between values num1 and num2 incremented each by one on each iteration. All iterations of such a loop are assumed to be independent.

3.4. DDM Function

```c
#pragma ddm func <name>
```

Defines a function that includes one or more DDM Codes-Block in its body.

3.5. User Defined Shared Variables

```c
#pragma ddm var <type> <name> <size>
```

Defines a variable as shared and allocates memory for it in the shared memory address space. name is the name of the variable, type shows the type of the variable and finally size, the number of elements to be allocated (useful for array structures).

3.6. System Configuration/Variables

```c
#pragma ddm kernelid <var>
```

Assigns the kernel ID to variable var.

```c
#pragma ddm kernel <number>
```

Defines the number of DDM Kernels in the program.

3.7. Debugging Primitives

```
#pragma ddmdebug print tsu <number>
#pragma ddmdebug print tsu all
#pragma ddmdebug print all
#pragma ddmdebug flush all
#pragma ddmdebug stats tsu <number>
```

These pragmas are related to debug functionality offered by the DDM-CMP simulator, such as printing the contents of the TSU.

4. Data-Driven Multithreading C Pre-Processor

The Data-Driven Multithreading C Pre-Processor (DDMCPP) is a tool that takes as input a regular C code program with the directives as described in the previous section and outputs a C program that includes all the library calls necessary for the program to execute on the DDM-CMP architecture. In addition, it embeds the Runtime Support System code. This tool is logically divided into two modules, the front-end and the back-end, which are described next.

4.1. DDMCPP Front-End

The DDMCPP front-end is a parser tool based on the `flex` and `bison` tools. The parser recognizes the directives presented in the previous section. The front-end is independent of the target architecture, i.e. its task is to parse the DDM directives and then pass the information to the back-end to produce the code corresponding to the target architecture.

4.2. DDMCPP Back-end

The back-end is built as the actions of the `bison` grammar for the DDM directives and is dependent on the target architecture. Currently we have a back-end for the DDM-CMP architecture but soon we expect to expand to more. The task of the back-end is to generate the code required for the DDM-CMP Runtime Support System such as the DDM-Kernel code, Thread select loop, and the load operations to the TSU, among others.

4.3. DDMCPP Usage

```
ddmcpp -K n [-debug]
   [-o outfile] infile
```

The pre-processor produces code for n kernels from the source code `infile`. Optionally the produced code may include debugging information (-debug) and the output, instead of being directed to the stdout may be directed to `outfile`.

5. Example of Code Transformation

In this section we describe the process of transforming the C example application depicted in Figure 4 into a DDM binary. This example program is very simple and it serves just for explanatory purposes. It is easy to identify that two set of independent instructions exist, `x++; y++;` (lines 03/04) and `x=x*z; y=y*z;` (lines 06/07).
01 main(){
02   int x=4; double y,z,k; y=8.1;
03   x++;
04   y++;
05   z=x+y;
06   x=z *x;
07   y=z *y;
08   k=x *y;
09   printf( "k=%g\n", k );
10 }

Figure 4. The original code of the example program.

Figure 5 shows the same code augmented with the necessary pre-processor directives to express the existing parallelism. Note that DDM threads normally contain more than just one instruction; here single-instruction threads are for explanation purposes only.

01 main(){
02   int x=4; double y,z,k; y=8.1;
03
04 #pragma ddm kernel 2
05
06 #pragma ddm block 1 \ 
07 import(int x,double y ) export( x:1, y:2 )
08
09 #pragma ddm thread 1 kernel 1 \ 
10 import ( x:0 ) export ( int x )
11 x++;
12 #pragma ddm endthread
13
14 #pragma ddm thread 2 kernel 2 \ 
15 import(y:0) export(double y)
16 y++;
17 #pragma ddm endthread
18
19 #pragma ddm endblock
20
21 z=x+y;
22
23 #pragma ddm block 2 \ 
24 import(int x,double y,double z) export(x:3, y:4)
25
26 #pragma ddm thread 3 kernel 1 \ 
27 import (x:0, z:0) export(int x)
28 x=z*x;
29 #pragma ddm endthread
30
31 #pragma ddm thread 4 kernel 2 \ 
32 import (y:0, z:0) export(double y)
33 y=z*y;
34 #pragma ddm endthread
35
36 #pragma ddm endblock
37
38 k=x*y;
39 printf( "k=%g\n", k );
40 }

Figure 5. The original code of the example program.

In line 04 the user defines that the program will be executed by 2 DDM-Kernels. Although a DDM Code-Block usually consists of more than just 3 threads, in this example we split the program in 2 such blocks again for better explanation.

The first block starts in line 06 with the `pragma ddm block` directive. This directive also defines the variables that will be used in its body, `x` and `y`, with the `import` part. The variables that are modified inside the block and are used in subsequent blocks are defined by the `export` part. This block ends at line 19 with the `pragma ddm endblock` directive. Similarly, in lines 23 and 36 we define the second block.

The first DDM thread of the program is defined to start in line 09 with the `pragma ddm thread` directive. The `import` part defines that variable `x` is used in the thread and is imported from the containing block (the zero after variable `x`). The `export` part defines that variable `x` must be exported by the DDM-Block as it is used by threads in the subsequent DDM-Blocks. Thread 1 ends with the `pragma ddm endthread` directive at line 12. Similar definitions exist for the other threads of the program.

Figure 6 depicts the Data-Flow Graph of the original program and its Data-Flow Graph after the preprocessing phase. Note that the DDM-Preprocessor has automatically added the required `Inlet` and `Outlet` threads.

Figure 7 depicts a snapshot the produced C code that, after being compiled with an ordinary compiler (e.g. gcc), can be executed by the DDM-CMP architecture (note that only a part of the program are shown for clarity issues).

Lines 09-16 regard the creation of the DDM-Kernels. Upon creation, the DDM-Kernels are redirected to `DDM BLOCK01` (line 49). There, each Kernel loads its TSU block directive. This directive also defines the variables that will be used in its body, `x` and `y`, with the `import` part. The variables that are modified inside the block and are used in subsequent blocks are defined by the `export` part. This block ends at line 19 with the `pragma ddm endblock` directive. Similarly, in lines 23 and 36 we define the second block.
with the *Inlet* thread of its first block and then redirects to the \texttt{THREAD\_SELECT} loop. In addition to loading the TSU with the block’s threads (lines 22 and 23 for the *Inlet* thread of the first DDM-Kernel for block 1), the *Inlet* threads copy the imported variables to a special structure, the \texttt{sharedResultsArray}. In line 28, the *Inlet* thread notifies the TSU about its completion and then redirects execution to the \texttt{THREAD\_SELECT} loop (line 29).

The first operation of execution threads, for example \texttt{THREAD1\_BLOCK1} (lines 31-37), is to import the necessary variables (line 32). Exported variables, are copied back to the \texttt{sharedResultsArray} structure. Similarly to the *Inlet* threads, after completing their execution, execution threads notify the TSU about this event (line 28) and redirect the execution to the \texttt{THREAD\_SELECT} loop.

The *Outlet* threads, first export the shared variables (lines 41-42), then notify the TSU about their completion and clear the allocated resources on the TSU (line 45). One of the *Outlet* threads, here \texttt{OUTLET\_THREAD\_BLOCK1\_KERNEL1}, executes the inter-block code, as such, this thread is redirected to \texttt{END\_BLOCK1} and not to the \texttt{THREAD\_SELECT} loop.

### 6. Future Directions

As mentioned before, the main advantage of a Pre-Processing tool is to ease the porting of applications to a new architecture. There is an obvious tradeoff between the simplicity of the tool and therefore the time to develop it and the ease of programmability. The tradeoff is that the easier the programmability the more complex the tool needs to be as we want to shift the responsibility from the user to the tool. In the case of the DDMCPP as presented in the previous section, we have left some responsibility for the user such as numbering the threads and blocks as well as defining the import and export variables.

One of the requirements of the current DDMCPP implementation that we are currently working on is to avoid the definition of the import and export variables. This should be done automatically in most cases and will leave the programmer to only define the boundaries of the threads.

In addition we would like to enhance DDMCPP with more features. One of the enhancements would be to accept in the front-end OpenMP code and transform it in order to produce DDM code as output. This will allow the users to port to DDM more applications, specially the ones that are already available for the OpenMP model. There is also the intend to perform the same for the MPI model.

### 7. Conclusions

In this paper we presented DDMCPP, a C Pre-Processor for the Data-Driven Multithreading model of execution.

---

**Figure 7.** The produced C code.
This tool offers two major benefits. First, it makes it easier to port applications to the DDM-CMP architecture as the original applications require only to be augmented with simple compiler directives. Second, it is a valuable tool to be used fast into the development of the new architecture and allows for the discovery of patterns in applications that will later on be recognized in a fully automatic compiler. Although such a compiler is the ultimate goal for every developer of a new architecture, the solution here presented is feasible therefore allowing for a better development and validation of the architecture as opposed to the most common hand-coded application development.

References


