Microthreading: model and compiler

Thomas Bernard[*,1], Chris R. Jesshope[*,1], Peter M.W. Knijnenburg[*,1]

[*] *Computer Systems Architecture, Informatics Institute - Universiteit van Amsterdam Kruislaan 403, 1098 SJ Amsterdam, The Netherlands*

ABSTRACT

There are two ways to improve processor performance, either by increasing the number of instructions issued per cycle or by increasing the speed of the processor's clock. However, the former increases circuit complexity for diminishing returns and the latter increases power dissipation. Our Microthreading model proposes an alternative approach to ILP based on code fragmentation. These code fragments are called microthreads and they can be run concurrently on a CMP chip. This concurrency is explicitly described in the source file by using a new language μTC based on the C language. The compiler produces specific code which contains new ISA instructions. Those instructions permit a family of microthreads to be created which can be a collection of loop iterations or heterogeneous processes. This paper presents an overview of the Microthreading model and the compilation chain targeting this new architecture.

KEYWORDS: microthreads; code fragments; chip-multiprocessors; compilation; compiler; front-end; back-end

1 Introduction

Nowadays, the latest generations of processors try to exploit multiple cores on a chip with 1, 2 or 4 processors, e.g. the Intel Core 2© processor family. Instead of looking for improvements based only on clock speed, other CPU features are being increased including cache size and the number of cores. Microgrids [MTArch05] present a CMP (section 3) with a virtually unlimited number of processors (it can be 10 or 1000 processors). The Microthreading model [CJM06] (section 2) proposes a different approach to ILP which is based on code fragmentation. The sequential code is broken down into code fragments, or microthreads, which are issued on processors in order to parallelize the computation. A specific new language (section 4) called μTC allows the programmer to develop concurrency-oriented applications. A dedicated compiler (section 5) produces microthreaded code for this new chip architecture.

[1]E-mail: {tbernard,jesshope,peterk}@science.uva.nl

2 The Microthreading model

Our Microthreading model aims to increase the concurrency of computations on the chip [CJM06]. Based on loop level parallelism, the code is split into fragments , i.e. microthreads, and those code fragments are executed on several processors as shown in Fig.1. A microthread is issued independently on a processor. The model adds just a few new instructions to an existing ISA to implement explicit concurrency controls and to define parametric sets of concurrent code fragments, which are scheduled dynamically on multiple processors, see section 3. Concurrency in Microthreading is parametric and its schedules are dynamic, thus allowing the same binary code to be run on an arbitrary number of processors. This allows the dynamic management of resources. These new ISA keywords [ILP05] also handle data dependencies between microthreads.

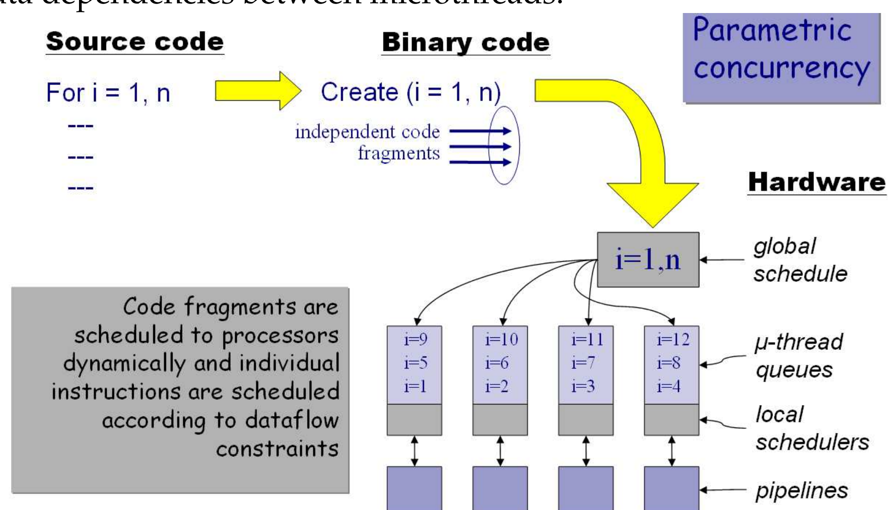

Figure 1: Overview of the model.

Our Microthreading model supports a shared-register model of data using blocking reads. The model also provides the programmer the opportunity to explicitly capture concurrency in the code by the means of a new language, as explained in section 4.

3 The Microthreaded architecture

A microthreaded architecture is a chip multiprocessor which in principle can comprise a very large number of processors. It does not matter how many processors execute the code, the same code will run on any number of processors bounded only by the loop limit. A global scheduler determines the distribution of the microthreads over the processors after receiving a family of microthreads (created by the ISA instruction **create**).

As shown in Fig.2, each processor consists of a continuation queue (stack of microthreads), a local scheduler which manages the microthread queue, a local pipeline which executes instructions, a local register file (memory for computations), links to the bus and a ring network (for communication with the others processors).

Each microthread can access a set of local registers ($L), a set of shared registers ($S and $D) which are both dynamically allocated, and global registers ($G) which can be accessed by all microthreads. Each microthread has its own set of local registers. Global registers contain global values which are accessible to all microthreads. Global values are derived from

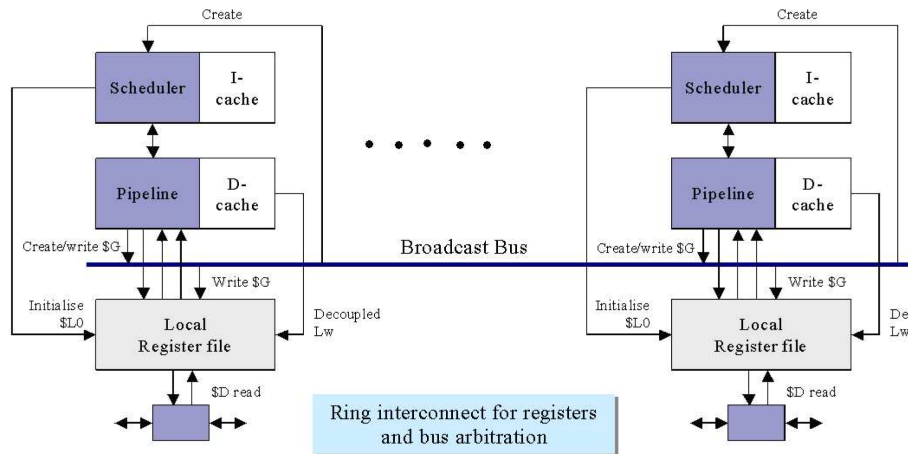

Figure 2: A microthreaded CMP.

local values in the creating thread and cannot be modified by the microthreads for maintaining determinism. The processors can exchange data in the shared registers for dependencies between microthreads by a ring network, as shown in Fig.2.

4 A new language: μTC

μTC is an extension of the C language. A set of new keywords and structures have been added to handle the microthreading model especially data dependencies and synchronization between microthreads. A μTC source file contains explicit statements to define the code fragments, namely the **create** statement which produces a family of microthreads. In the example below, each iteration is a microthread which can be issued on a processor.

```
create (i_fid; 0;m-1)
{
     index int i;
     sum[i]=i+1;
}
sync (i_fid);
```

For handling synchronization, the **sync** statement blocks until the family of microthreads specified by *i_fid* has completed and then completes its execution. This example has the same behaviour as a **for** loop statement with $m-1$ iterations, but here each iteration is executed concurrently. The programmer can also explicitly specify structures which can be microthreaded by means of the **thread** function specifier as follow.

```
thread sumint(shared int sum, int array[])
{
     index int idx;
     sum = sum + array[idx];
}

int main()
```

```
{
      int *a;
      int fid, s=0, n=10;
      create(fid; 0; n-1) sumint(s, *a);
      sync(fid);
}
```

The **shared** keyword is a type qualifier and notifies the compiler of a dependency of variables between iterations. Finally, **index** is another type qualifier which represents an index within the family of microthreads called *fid*.

5 μT compiler

We are designing a core compiler which takes as input a μTC source file and produces microthread binaries as output. The μTC front-end handles the new concurrent language. It checks for lexical, syntax and also semantic errors. The Intermediate Representation (IR) of the compiler keeps the microthreaded information. Optimizations are performed at the middle-end level. The back-end produces the microthreaded code by using the new ISA keywords. [MTArch05] discusses the back-end code generation for microthreads and high-level code transformations which can be done by the core compiler.

6 Conclusion

In this paper, we have discussed a novel approach to ILP based on code fragmentation, called microthreading. A microthreaded architecture consists of many simple in-order pipelines that can execute these code fragments in parallel. Currently, we are implementing a core compiler for μTC and a set of tools for our architecture. This set of tools consists of source-to-source compilers (μTC to C and C to μTC). We are also extending the μTC language. After finishing these tools, many experiments will test the quality of code, the performance, use of memory, etc.

References

[MTArch05] T. Bernard, K. Bousias, B. de Geus, M. Lankamp, L. Zhang, A. Pimentel, P.M.W. Knijnenburg, and C.R. Jesshope. (2006) *A Microthreaded Architecture and its Compiler*, Proc. 12th International Workshop on Compilers for Parallel Computers (CPC), M. Arenez, R. Doallo, B.B. Fraguela, and J. Tourino (eds.), pp 326-340.

[CJM06] C.R. Jesshope. (2006) *Microthreading - a distributed paradigm for instruction-level concurrency*, to be published, Parallel processing Letters - Proc. of IFIP 10.3 Workshop 2003.

[ILP05] K. Bousias, N.M. Hasasneh and C.R. Jesshope. (2006) *Instruction-level parallelism through Microthreading - a scalable Approach to chip multiprocessors*, Computer Journal, 49 (2), pp 211-233.