Morphable Multithreaded Memory Tiles (M3T) Architecture

Jose Renau, **James Tuck**[†], **Wei Liu**, and **Josep Torrellas**
Department of Computer Science
[†]Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign
http://iacoma.cs.uiuc.edu/m3t
UIUC-CS Technical Report, September 2002

Abstract

A M3T chip is composed of many general-purpose RISC cores interleaved with memory blocks. Cores and memories are reconfigured at run time, allowing the chip to morph into a TaskScalar, VLIW, or MIMD engine, or even a combination of them. TaskScalar is a mode of execution that supports very fine grain multi-threading with thread-level speculation. The result of M3T polymorphism is the highest-performing, most energy-efficient solution at each stage of the computation. M3T's polymorphism is enabled by several special microarchitectural supports provided in the chip that provide efficient support for core-to-core synchronization, core-to-core data communication, and core-to-core control communication. M3T also has a novel compiler that reconfigures the platform, both the hardware and the morphware.

1 Overview

M3T supports polymorphism through the addition of hardware specifically designed to benefit different execution models or *architectural templates*. A M3T chip is composed of many general-purpose RISC cores interleaved with memory blocks. By default, the Sequential and MIMD execution models perform well on this architecture. Through the addition of special hardware, M3T also supports the TaskScalar and VLIW templates, thus providing polymorphism. TaskScalar is a threading mode of execution that supports very fine-grain tasks and thread-level speculation. At run time, and under application command, the M3T chip can morph into one of these architectural templates, or into several of these templates sharing the chip.

1.1 Microarchitecture

M3T's ability to dynamically morph is enabled by several novel combinations of microarchitectural supports present in the chip. These supports are core-to-core synchronization, core-to-core data communication, and core-to-core control communication. A M3T chip can synchronize all cores (or multiple groups of them) in a handful of cycles. M3T enables core-to-core data communication by allowing a core to read or write a register-like structure in another core. Core-to-core control communication is supported by allowing a core to broadly control the datapath of another one.

With these supports, the different architectural templates are implemented as follows:

MIMD: Each processor operates independently, executing its own stream of instructions. Communication occurs only through reading and writing words in memory. Of our three supports, only the synchronization support is used at all. This template enables efficient execution of coarse-grain explicitly parallel code.

TaskScalar: The instruction stream is broken down by the compiler into very fine-grain tasks. These tasks are dynamically assigned to the cores. A task assigned to a core may not be eligible to execute because of a data or control dependence with a predecessor task that has not completed. When such a dependence is resolved, the task is allowed to execute. This mode of execution requires the ability of a core to update the register structures of another (e.g. when passing the input arguments in a task spawn), control another core (e.g. stall that other core until the dependence is resolved), and quickly synchronize, if necessary. Overall, this template enables aggressive parallelization of irregular codes.

VLIW: Each core executes its own stream of instructions but the cores operate in lockstep. The union of the independent streams is equivalent to a single VLIW stream. This template works best when the streams of the different cores are very similar to one another. Cores synchronize every few instructions. In each step, at most one operation can be a branch. If the branch is predicted taken, the processor executing the branch uses core-to-core control communication to force all the other processors to jump to the new VLIW word. Furthermore, data may also be passed from one core to the registers of another. Finally, our support for very fast synchronization ensures that all processors execute in lockstep.

Table 1 shows the primitives used by each template. For a given piece of software, the programmer or compiler can select among these templates. The result is the highest-performing, most energy-efficient solution in each section of the application.

Architectural Template	Core-to-Core Synchronization	Core-to-Core Data Communication	Core-to-Core Control Communication
MIMD	X		
TaskScalar	X	X	X
VLIW	X	X	X

Table 1: Microarchitectural supports required by each architectural template.

1.2 Compilation

The M3T polymorphous compiler is a source-to-source C compiler responsible for generating code that will be executed on different architectural templates. Because automatic selection of templates is difficult, the compiler needs hints to decide on which template a particular kernel should execute. Rather than extend the language with special constructs, the compiler relies on pragmas, either inserted by the programmer or generated automatically by synthesis tools, to select the appropriate architectural template and generate the right code.

The pragmas employed in the source code provide a variety of information to the compiler. *Architectural reconfiguration* pragmas indicate how the architecture should be reconfigured at runtime. *Code break up* pragmas indicate how the program can be broken into modules. Depending on which architectural template it is chosen, the compiler optimizes the code's data structures and paths differently. Finally, *module scheduling* pragmas indicate how, when, and where the module should be scheduled. These pragmas may specify that a module be executed under certain conditions, or they may identify synchronization and communication between one module and others. There may also be times when the nature of the application is unknown at compile time. In these cases, pragmas may be used to specify alternative architectural templates.

2 Templates

2.1 TaskScalar

Parallel programs can utilize the MIMD architectural template in M3T. However, the TaskScalar template is designed for situations when the application does not have enough coarse-grain parallelism, and using fine-grain tasks results in too much overhead. For these situations, the TaskScalar template provides fast synchronization and communication between cores. Special hardware glues the cores, resulting in a tightly-coupled architecture that executes very small tasks.

Cores collaborate in building enough task-level parallelism. To enforce dependences between tasks, the TaskScalar uses the Pending Task Queue (PTQ). The PTQ has an entry for each active task. Also, the PTQ is not a centralized structure; it is distributed between cores. The PTQ checks for dependences between tasks and enforces them in a way similar to thread-level speculation.

2.2 MIMD

The MIMD execution model directly benefits from the underlying M3T multiprocessor architecture. Parallel applications already developed using a MIMD philosophy will run well on M3T. Furthermore, since all the cores are located on a single

chip, communication between threads is less costly than in a multichip implementation.

2.3 VLIW

There are many scientific applications which have very little task level parallelism but exhibit large instruction level parallelism. If these applications were forced to run sequentially on a chip multiprocessor with small cores, the performance would be very poor. In large applications, this sequential code would limit the performance of the entire application significantly.

VLIW architectures can execute these codes well because loops can be pipelined and scheduled very tightly across all the functional units. In M3T, we distribute a small piece of code across multiple cores in order to utilize the hardware in the same way a VLIW does. Each core computes part of the computation and synchronizes at set intervals with its neighboring cores. This results in a multithreaded lock-step execution of the program.

The VLIW template uses our synchronization support to synchronize in a handful of cycles. This provides a tightly coupled execution across cores. As a result, we are able to parallelize loops and other structures with a large number of dependences without suffering the cost of expensive synchronization operations.

3 Status

We are currently completing the simulator that will enable us to evaluate the potential of dynamically morphing the hardware as an application runs. We will report on the results very soon.