Sun Microsystems Laboratories

Subject: TPE: A network of closely coupled computational elements
Date: September 23, 2008
Authors: Igor Benko, Mark Greenstreet
SML#: 2008-443

THE PROBLEM

Among major challenges facing processor designers today are increasing performance while keeping power consumption under control and decreasing design cost. To address these challenges, we propose a new processor architecture that we call Tiny Processing Elements (TPE). A distinct feature of the TPE architecture is that it is constructed of clusters of incomplete "processors" that we call TPEs. TPEs run in parallel and are intended to operate collaboratively, hence a TPE can be significantly simpler than a complete processor core.

TPE architecture builds on ideas from our earlier work on FLEET [22]. Both TPE and FLEET architectures put a programmer in control of data movement inside a processor and both assume asynchronous data communication. The TPE architecture and FLEET, however, take fundamentally different approaches to performing computation: FLEET is an asynchronous transport-triggered architecture [23] with significant enhancements in programmability of data movement. Computation in FLEET takes place as a consequence of data being moved to an appropriate functional unit called a ``ship.'' These ships have limited programmability and simply react to compute new results in response to receiving new operands. The TPE architecture, on the other hand, combines programmability of data movement with programmability of functional units, and adds the ability of functional units to pick for each other what code block to execute. A result is an unusual paradigm where code execution takes place as a collaboration of concurrently executing individual code blocks. Individual functional units in the TPE architecture have greater capabilities than individual ships in FLEET, but are still much smaller than the "cores" of today's chip multiprocessors.

TPE ARCHITECTURE OVERVIEW

A TPE cluster is the fundamental unit of computation in the TPE architecture. A cluster consists of a network of Tiny Processing Elements (TPEs) which execute a program together.

The distinguishing feature of a TPE cluster is software control of hardware-level parallelism. Each program that executes on a TPE cluster executes as a collection of short execution sequences that run on a subset of the TPEs comprising the cluster. Because TPEs operate in parallel and are intended to operate cooperatively, each TPE can be significantly simpler than a fully functional processor core.

TPEs within a cluster are connected by an order-preserving switch fabric, and software controls communication between TPEs. All inter-TPE communication is asynchronous. The cost of communication between TPEs is non-uniform. We expect that communication between neighboring TPEs will be very fast.

The base unit of program execution is a code block. A code block is a short sequence of instructions and executes on a single TPE. TPEs within a cluster execute code blocks in parallel. Moreover, TPEs can influence each other by telling each other which code to execute next.

TPEs in a cluster have different capabilities, so a TPE network is always heterogeneous. At the very least, a subset of the TPEs perform arithmetic operations, and a subset of TPEs provide an interface to memory. Note that TPEs may also differ in size. For example, a TPE that performs floating point arithmetic will be larger than a TPE that performs only integer operations. Also note that TPE may differ in their relative speed and power consumption. Despite TPE heterogeneity, several TPEs within a cluster might be identical.

BACKGROUND

Several projects have either proposed or implemented networks of processing elements. [3,4,5,6,7,9, 21] refer to examples of such undertakings where basic building blocks are simple, albeit full scale processors, most often versions or derivations of MIPS processors. Another class of related work are systolic arrays [15], which implement a limited computation paradigm. Graphics processors such as NVIDIA Tesla, and extreme multithreaded processors such as IBM Cyclops [16] contain many processing units that communicate through common memory instead of directly to each other. Dataflow architectures [18, 19, 20] may contain several processing units. The original dataflow paradigm [18,19] targets parallelism at the instruction level by introducing hardware for matching available data with instructions waiting for operands. Hybrid dataflow architectures [20] extend that paradigm to parallelism among code blocks, but retain the notion of matching available data/operands with code blocks.

A TPE

TPE clusters comprise of a network of TPEs (Tiny Processing Elements). A TPE has a limited computation capability. For example, a TPE may be able to perform only simple integer operations but might have no capability to access memory. To gain memory access, such a TPE cooperates with TPEs whose core functionality is to pass data between TPEs and the memory.

A TPE contains a functional unit that performs its core function. Additionally, a TPE contains a small number (expected 8 to 16) of registers which are used as local data storage. A small number (expected 4) of input registers with FIFO semantics are used for receiving data from other TPEs. Note that input registers can be used as inputs to instructions performed by the TPE. Reading a value from an input register removes the value from the input FIFO. The read is blocking. That is, the instruction blocks until data appears in the input register.

A TPE also has a number of output registers used to send data to other TPEs. Each output register is paired with the destination TPE address register, and a destination index register that determines which input register at the destination TPE will receive the data. Once the destination address and index are set, writing into an output register causes data to be sent to the specified destination. Moreover, the destination address and index can be reused to send multiple data items from one TPE to another.

A TPE is programmable and executes a simple instruction set. A subset of instructions is specific to each type of a TPE. These are the instructions that engage the TPE's specific functional unit. The remaining instructions are common among TPEs. These instructions encompass moving data between registers, communication between TPEs, code block loading, and conditional execution.

TPEs have an unusual code loading and code execution paradigm. The code is organized into code blocks, which each consist of a small number, a dozen or so, TPE instructions. Each TPE contains a small code block register file, where each register stores a code block. The size of a code block register is the same or a small multiple of the size of a cache line.

There are two kinds of code block registers. Local code block registers serve as local storage for small number of code blocks. Code execution, however, takes place from active code block registers. Before a code block is executed, it is copied from a local code block register to an active code block register.

Local code block registers act as input registers with FIFO semantics. A TPE can perform a peek operation on a local code block register, which acts as a non-destructive read and does not remove an item from the FIFO. Both read and peek operations block TPE execution until an item arrives in the corresponding queue.

The contents of code block registers arrives from other TPEs. The actual source of a code block may be memory; such a code block would arrive via a code-block loader TPE that provides interface to memory. That is, a TPE can request that a code-block loader TPE deposit a certain code block into a specified local code block register. Moreover, a TPEs can ask a code block loader to deposit a code block in a specific local code block register in another TPE. Such ability to influence what code blocks are executed by other TPEs is essential to have a cluster of TPEs operate in concert.

The source of a code block may also be a TPE, which stores that code block and, when appropriate, sends it to a destination TPE. Such caching of code blocks enables fast execution of conditional code. Namely, the source TPE stores code blocks for alternative paths of execution. When the execution path is determined, the source TPE sends an appropriate code block to the destination TPE.

Upon startup, a TPE executes the code block in local code block register zero. Because of FIFO semantics for local code block registers, a TPE blocks until its first code block is delivered. Then the content of the local code block register zero is moved to the active code block register 0 and that code is executed. Note that a TPE cluster requires a boot mechanism that loads code blocks that start execution of the cluster. A simple such mechanism can be built into a code-block loader TPE, which would read a boot code block from a specific location.

A TPE has the ability to execute conditional code. Similar to the traditional jump instruction is a jump to a location within the active code block registers. The second version of the jump instruction copies or moves the contents of a local code block register to an active code block register and then executes that code block. Note that the copy or move operation will block if the local code block register is empty. We use that blocking to have other TPEs determine what code executes next and when the execution begins. Note also that a TPE can request a new code block from the code block register loader. Such a request can take place well before the code block is needed, which means that we can pipeline code load along with code execution.

A TPE CLUSTER

A TPE always operates within a TPE cluster where TPEs are connected with one or more networks. We expect that communication between neighboring TPEs will be very fast. Another important property of a TPE cluster is the selection of TPEs that comprise a cluster and geographic location of different TPEs. While further research is required before we can be very specific about TPE cluster composition, we can safely say that a cluster will comprise of a combination of the following:

- Compute/ALU TPEs that perform integer and/or floating point computation, or a specialized function such as cryptography operations.
- Memory access TPEs. These TPEs have enough computation power to perform basic address arithmetic, such as keeping track of a counter and a stride when being asked to store or to read a sequence of values. Note that we may want to separate memory access TPEs that deal with data for the computation, and code-block loading TPEs that supply code blocks to other TPEs.
- Routing TPEs that implement routing algorithms and help with moving data and instructions across a TPE cluster and between clusters.
- Communication TPEs that provide connections between TPE clusters and that provide access to other external communication such as networking. Note that a subset of TPE clusters in a TPE processor chip may be tasked with doing nothing but providing computation required for networking.

An inter-TPE network provides an asynchronous communication fabric of finite capacity. To prevent the network from clogging, TPEs provide a simple flow-control mechanism. Each output register is equipped with a credit counter specifying how many data items can be in flight to the specified destination. Sending a data item decrements the credit counter. When the destination TPE receives a data item, it sends an acknowledgment back to the sending TPE, which, in turn, increments the credit counter.

Our limited programming experiments to date suggest that computation in a TPE cluster can be organized in such a way that most data communication takes place between neighboring TPEs. Note that the topology of a TPE cluster presents a new dimension for compiler optimization. Namely, a compiler will be tasked to balance communication cost with other performance parameters, thus opening the door for different optimization targets, some biased towards speed and some biased towards low energy consumption. Note that ability to influence energy consumption by selecting communication patterns and by selecting what functional units will perform certain parts of a computation is inherited from transport triggered architectures [23] and FLEET [22].

Subroutine calls, traps, and interrupts are important factors in processor design and architecture. For the TPE architecture, these operations will have to be defined at the level of a TPE cluster. While we see several options for implementing these mechanisms in a TPE cluster, we require further modeling and experimentation to settle on the approaches we would take.

A TPE PROCESSOR

A TPE processor consists of several TPE clusters. Drawing a parallel between a TPE processor and a chip multiprocessor (CMP), we can also draw parallel between a single TPE cluster and a core in a CMP. One major difference is that in commercial CMPs, cores communicate through shared caches. In the TPE architecture, on the other hand, we assume an inter-cluster network that enables communication between TPEs. Another major difference is that a commercial CMP core is a monolithic fully functional processor while a TPE cluster consists of a group of simple TPEs, none of which alone has the functionality of a full processor core.

Each TPE cluster has a private L1 cache or local memory that connects it to the main memory. The cache stores both data and instructions, possibly in separate banks. Note that we do not assume coherency between L1 caches.

PROGRAMMING A TPE CLUSTER

One of key intentions behind the TPE architecture was to provide new mechanisms for control that a program has over its own execution. Fundamentally, the TPE architecture provides direct means for interaction between “strands” and “threads” of execution. Moreover, programmability of and explicit communication with “memory-access TPEs” opens the door for adjusting memory access and cache management in a dynamic manner, depending on the need of an application. Our early investigations [24, 25, 27] indicate a potential for instruction-level parallelism in TPEs to rise well beyond today's limit of less than two.

Multitude and differentiation of TPEs, ability of TPEs to exchange data, and ability that a TPE influences what code block are executed on another TPEs are key properties of a TPE cluster that influence programming techniques for TPEs. We refer the reader to [24, 25, 26, 27] for specifics of our early experiments with TPE programming. In this memo, we limit ourselves to general remarks and observations:

- TPE programs consist of a collection of code blocks and of assignments of code block execution on different TPEs.
- Each code block is inherently sequential, but may involve communication with other TPEs.
- Communication between TPEs is asynchronous from a programmer perspective. The actual implementation of TPEs and/or inter-TPE networks can be done either with synchronous or asynchronous circuits, or a combination.
- A program consisting of a group of code blocks is inherently parallel: A subset of code blocks execute in parallel, and a schedule of code block execution is dynamic and is determined during execution of the program.
- Conditional execution can be implemented through interaction between TPEs. One TPE computes the conditions and then triggers execution of appropriate code block in one or more TPEs.

- Code block execution connects TPEs as pipelines or other more complex arrangements. These arrangements are dynamically set up and torn down during program execution. Note that these structures are defined by communication patterns and not by the physical location of TPEs.

- Parallel execution of code blocks can be compared to a combination of more traditional instruction level parallelism and thread level parallelism. Parallel execution of code blocks and individual instructions to code blocks is more general and more powerful than instruction-level parallelism. We achieve thread-level parallelism by assigning each thread a subset of TPEs within a cluster; these TPEs then execute the code that is equivalent to single-thread execution.

- TPEs within a cluster can be partitioned dynamically into independent groups that each performs an independent set of operations. For example, multiple threads can execute on one TPE cluster at the same time. Furthermore, these threads have inter-TPE communication available to them, so inter-thread interaction can be light-weight.

- A subset of TPEs can be allocated to perform auxiliary functions such as scouting, or cache management. In TPEs these functions can be defined by software instead of being hardwired.

- Known execution concepts such as pipelining, superscalar execution, and VLIW can all be achieved on TPEs. TPEs can combine these execution modes dynamically, depending on properties of the execution.

- By storing the same piece of data (e.g. a variable for counting loop iterations) on multiple TPEs we may achieve higher performance because we eliminate the need to communicate the data between TPEs and synchronize TPE execution. Note that this technique does not apply to traditional processors. There, this technique would compare to storing the same variable in multiple registers within a single CPU and performing the same operation on all of the registers, clearly with a negative impact on execution time.

CONCLUSIONS

The following five ideas are key to TPE architecture:

- TPEs specialized and none implements the complete functionality of a processor core. Parallel and cooperative operation of TPEs within a cluster makes specialization possible. Consequently, we expect individual TPEs to be very small and significantly simpler today's processor cores.

- Blocking asynchronous data communication between TPEs opens the door to heterogeneity of TPEs and decouples execution of code blocks on different TPEs. As a consequence, communication patterns within a TPE cluster change dynamically, and TPE programming techniques can build on top of known message-passing paradigm.

- Communication of code blocks is also blocking and asynchronous. Code blocks can be passed between individual TPEs, so code executed on a TPE may arrive from another TPE and not from a central fetch unit.
- TPEs can pick for each other what code block will execute next. Along with the ability to store code blocks locally, this feature enables fast conditional branching by applying simple techniques for speculative code fetching.
- A subset of TPEs within a cluster can be used to perform auxiliary functions such as scouting, processor monitoring, or measuring performance for execution of code traces. Such auxiliary functions are now implemented in software, greatly simplifying the hardware making it possible to deploy auxiliary functions on the basis of specific application needs.

References

[1] A Communication Centric CPU Proposal, by Mark Greenstreet and Igor Benko, SML# 2007-0290, July 2007

[2]Introducing Tiny Processing Elements (TPE), by Igor Benko, SML# 2008-0133, April 2008

[3]The Raw Microprocessor: A Computational Fabric for Software Circuits and General Purpose Programs, by Michael Bedford et al, IEEE Micro, Mar/Apr 2002

[4] Space-Time Scheduling of Instruction-Level Parallelism on a RAW Machine, by Walter Lee et al, Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems(ASPLOS-VIII), San Jose, CA, October 1998

[5] Exploiting Software Pipelining for Network-on-Chip architectures, by Feihui Li, Mahmut Kandemir, and Ibrahim Kolcu, Proceedings of the 2006 Emerging VLSI Technologies and Architectures (ISVLSI'06), 2006

[6] On-chip Interconnection Architecture of the Tile Processor, by DavidWentzlaf at al, IEEE Micro Sep/Oct 2007

[7] Scaling to the End of Silicon with EDGE Architectures, by D. Burger et al, IEEE Computer, 37 (7), pp. 44-55, July 2004

[8] May, D. 1988. The influence of VLSI technology on computer architecture. Supercomputing 88, International Supercomputing Institute, St. Petersburg, FL, 247–256. (machine instructions for process creation, communication and termination)

[9] Implementing Tile-based Chip Multiprocessors with GALS Clocking Styles, by Zhiyi Yu and Bevan Baas, International Conference on Computer Design, Oct 2007

[10] A Chip Mulithreaded Processor for Network-facing Workloads, by Sanjiv Kapil et al, IEEE Micro 2004

[11] Next Generation Intel® Micro-architecture (Nehalem) Clocking Architecture, Nasser Kurt et al, IEEE Symposium on VLSI Circuits, June 2008

[12] AMD multicore architecture, http://multicore.amd.com/us-en/AMD-Multi-Core.aspx

[13] Cell processor Low Power Design Methodology, by Daniel Stasiak et al, IEEE Micro 2006, volume 2

[14] John Hennessy and David Patterson, Computer Architecture: A Quantitative Approach, Third Edition, Morgan Kaufmann Publishers, Inc. 2003

[15] VLSI Array Processors, by S.Y. Kung, Prentice Hall 1988

[16] Dissecting Cyclops: A Detailed Analysis of a Multithreaded Architecture, by George Almasi et al, SIGARCH Computer Architecture News, http://www.research.ibm.com/people/c/cascaval/medea02.ps

[17] NVIDIA Tesla: A Unified Graphics and Computing Architecture, by Erik Lindholm et al, IEEE Micro, March-April 2008

[18] Executing a Program on the MIT Tagged-Token Dataflow Architecture, by K. Arvind and R.S. Nikhil, IEEE Transactions on Computers, March 1990

[19] Monsoon: An Explicit Token Store Architecture, by Greg Papadopoulos and David Culler, International Symposium on Computer Architecture, 1990

[20] Toward a dataflow/von Neumann hybrid architecture, by Robert Iannucci, ACM SIGARCH Computer Architecture News, Vol. 16, pp 131-140, 1988

[21] Transputer Hardware and System Design, by J. Hinton and A. Pinder, Prentice Hall, 1993

[22] FLEET architecture, http://fleet.cs.berkeley.edu/

[23]Microprocessor architecture: From VLIW to TTA, by Henk Corporaal, Wiley, 1998

[24] TPE Sorting and Simulation, by Brad Bingham, SML #2008-0357, July 2008

[25] Performance via Redundancy in TPEs, by Brad Bingham and Igor Benko, SML# 2008-0384, July 2008

[26] TPE Codeblock Transition and Reset, by Mark Greenstreet, Igor Benko, and Brad Bingham, SML# 2008-0414, August 2008

[27] Code Snippet Compilation Examples for TPEs, by Brad Bingham and Igor Benko, SML# 2008-ib16, August 2008

[28] IBM Power6 Microarchitecture, http://researchweb.watson.ibm.com/journal/rd/516/le.html

[29] An 80-Tile 1.28TFLOPS Network-on-Chip in 65nm CMOS, by Vangal et al, ISSCC 2007

[30] VLIW: Failure or Lesson, by Max Baron, Microprocessor Review Editorial, June 30, 2008