Retrospective: Exploiting ILP, TLP, and DLP with the Polymorphous TRIPS Architecture

Karthikeyan Sankaralingam[†,⋆] Stephen W. Keckler[†] Doug Burger[‡]
[†]NVIDIA [⋆]University of Wisconsin-Madison [‡]Microsoft

I. Historical Context

When this paper was published in 2003, CPUs were the dominant computing platform but faced challenges to single-threaded performance scaling due to limitations on clock rate [1] and power [5]. Application-specific processors targeting desktop, server, network, scientific, graphics, and digital signal processing workloads were coming to the marketplace as alternatives to fully general purpose CPUs. One differentiator among these types of processors is the granularity of parallelism supported in the architecture. This ranges from many fine grained processing elements as proposed by a range of reconfigurable ALU array architectures, to a few very powerful and large out-of-order processing cores. At that time, short vector extensions were just finding their way into general-purpose CPUs and GPUs supported only programmable graphics with low-level programming languages such as Cg. The challenge for these specialized architectures was one of *design fragility*, in which an architecture tailored for one type of workload would perform poorly when processing a different type of workload. The polymorphous computing paradigm was emerging as a way to address fragility to run different workloads on one hardware.

This paper investigated a type of polymorphous computing which allowed a single hardware substrate to support different granularities of parallelism to support various workloads - instruction-level parallelism (ILP), threaded-parallelism (TLP), and streaming data-level parallelism (DLP). The paper explored support for polymorphism in the ISA (and hence the compiler), and the implementation of polymorphic hardware mechanisms designed to support multiple types of parallelism in the compute substrate and memory hierarchy. It also examined two methods of designing polymorphous architectures that aim to support multiple granularities of parallelism: (1) synthesis in which small processing elements can be aggregated to form a larger processor, and (2) partitioning in which a large processor is decomposed into many processors.

II. The Polymorphous TRIPS Architecture

The paper proposed the TRIPS architecture, which used a polymorphous computing substrate with an adaptive, polymorphous on-chip memory. We constructed a high-performance ILP processor with a CGRA execution subtrate, blending principles of speculative execution and distributed fine-grained processing to create one large logical core - a 16-wide out-of-order issue processor with a 1K instruction window. Thread and data parallelism were supported on the CGRA substrate using different control mechanisms and partitioning. We showed that this partitioning approach results in a less fragile architecture by using polymorphous mechanisms to yield high performance for both coarse and fine-grained concurrence.

ISA. The TRIPS architecture employed a block-oriented ISA which explicitly encoded predicated dataflow dependences between operations within a block and executed blocks atomically to facilitate block-level speculation. Blocks could be mapped to different sized collections of hardware units to facilitate execution at both fine and coarse granularities. The block-oriented ISA exposed more explicit concurrency than sequential ISAs, supporting higher degrees of ILP and DLP within a single ISA.

Polymorphous hardware. TRIPS polymorphous hardware mechanisms included a *frame space* for in-flight inter-instruction values, *register file banks*, *block sequencing control*, and a configurable *memory system*. The framespace was akin to reservation stations, but exposed through the ISA, allowing different modes of parallelism to use memory differently. The TRIPS memory tiles could be configured to behave as NUCA style L2 cache banks [2], scratchpad memory, or synchronization buffers for producer/consumer communication. In addition, the memory tiles closest to each processing tile present a special high-bandwidth interface that enabled them to be used as streaming register files.

Compiler. We examined one approach to polymorphism in which the programmer specified the type of parallelism (ILP, TLP, or DLP) a block was targeting. Based on the block's ISA encoding, the hardware's polymorphous resources behave differently, such as using the framespace for multiple threads versus speculative execution. The project also developed compiler techniques to handle spatial mappings for the CGRA.

III. Reflection and Perspective

A. Were all three forms of parallelism important?

Perhaps not surprisingly, all three forms of parallelism have continued to grow in importance, albeit at different rates. While increases in ILP have slowed, single-thread performance continues to be important. Instead of doubling every 18 months per early Moore's law curves, single-thread performance has improved at a rate of about 6% per year from the Pentium-4 processor (state-of-the art in 2003) to a contemporary Apple M1 processor running at a slower clock rate. The improvements have come largely from wider execution and more efficient speculation, representing "specialization" targeting

single-threaded applications, exceeding the raw ILP we could obtain in TRIPS. In retrospect, given that area became cheaper relative to power, polymorphism benefits we provided didn't outweigh it's costs.

In contrast, data-level parallelism has exploded in importance, including vector extensions to CPUs, SIMT execution in GPUs, tensor accelerators, and a panoply of domain-specific architectures. In particular, the rise of practical machine learning algorithms (which in turn depend on both high-performance data parallel architectures and massive training data sets) has driven many new and differentiated approaches, including architectures with explicit tensor instruction sets, wafer-scale integration, specialized blocks in FPGAs, and machine-learning datacenter architectures. Some of these architectures employ explicit dataflow execution models akin to the TRIPS principles for compute substrates.

Thread-level parallelism has bifurcated in two ways. Driven in part by cloud computing demands, datacenter processor core counts have increased to facilitate independent processes to execute on independent cores, supporting 2 to 8 threads per core. This form of TLP (or more specifically task-level parallelism) involves architects determining the balance of area devoted to core, cache, and memory-controllers at design time. In contrast, traditional GPU-like architectures employ massive multithreading both within and across processing cores, effectively providing "thousand-way" threaded parallelism for a single application. While this parallelism can be used in support of regular DLP, it can also provide parallelism to divergent multi-threaded workloads. Kolodny et al. have analyzed this multi-core and many-core tradeoff [3].

B. How important is a unified ISA?

While our paper argued for a single unified ISA for all three forms of parallelism, modern processors generally employ different ISAs to target ILP and DLP, particularly in the context of domain-specific processors. However, this line is somewhat blurred for programmable processors due to the rise of embedded accelerator hardware. For example, both CPUs and GPUs now employ specialized tensor accelerators, such as Intel's AMX matrix extensions and NVIDIA's Tensor Cores, that are tightly coupled to the general purpose execution cores.

While embedding hardware accelerator units within a programmable processor addresses one aspect of fragility, programming challenge is addressed elsewhere. Today, framework layers such as PyTorch, Tensorflow and hardware specific middleware such as TensorRT then automatically translate and optimize the algorithms for specific hardware platforms. This approach improves portability across different types of architectures (CPU, GPU, and domain-specific), while reducing system level fragility by allowing a single program expression to run in different ways on different hardware. However there is a growing complexity in developing these software frameworks, especially with growing heterogeneity in the hardware. In retrospect, the specialization problem we were attempting to solve in the ISA, is now being attacked in a software middleware abstraction.

C. Did polymorphism turn out to be important?

Compute units. Specialization has clearly been preferred to polymorphism for compute units in mainstream computing platforms. The rise of large domains such as AI has instigated not only domain specific architectures such as Google's TPU but also the embedded tensor accelerators described earlier. Performance, energy, and area efficiencies provided by hardware designed for a narrow task has superseded the capabilities of configurable hardware. In the area of granularity, both large (e.g. systolic arrays in TPUs) and moderately sized compute engines (e.g. Tensor Cores in GPUs) have been fruitfully deployed. For datacenters, we have seen a trifurcation with modern systems being optimized for (1) high-single thread performance (small number of large cores), (2) high multi-thread performance (many small cores), or (3) and large DLP (modest core + one or more GPUs per system).

Memory hierarchy. We would argue that polymorphism has become commonplace in processor memory systems. Many modern processor now employ common on-chip SRAM for different purposes including caches and software-managed scratchpads as we suggested in our 2003 paper. In some cases, this hardware is configurable (such as the split between caches and scratchpad in NVIDIA GPUs), while in others it is programmatic with memory operations that influence replacement policies to enable software-based cache control. We expect this trend to continue with additional memory-based capabilities such as queues and stream buffers.

This paper addressed the problem of fragility — the inability of processors to handle different classes of parallelism. That problem turned out to be important, using approaches different than the focus of this paper. Application frameworks have allowed hardware specialization to be the preferred path to extract high energy efficiency, while mitigating to some degree—programmability and productivity challenges. The problems our spatial compiler addressed are now commonly encountered in production systems, and solved using frameworks like integer linear programming our work inspired [4], and likely to grow in importance. Some of the underlying mechanisms we proposed for distributed compute processing and polymorphous memories have appeared in many systems in practice today. Given the hard limits of energy efficiency, we expect this specialization trend to continue. Continued work is needed to balance programmability and computational efficiency, which this paper tried to address.

REFERENCES

[1] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger, "Clock Rate versus IPC: The End of the Road for Conventional Microarchitectures," in *ISCA*, 2000.

[2] C. Kim, D. Burger, and S. W. Keckler, "An Adaptive, Non-Uniform Cache Structure for Wire-Delay Dominated on-Chip Caches," in *ASPLOS*, 2002.

[3] A. Kolodny, E. Bolotin, I. Keidar, Z. Guz, A. Mendelson, and U. C. Weiser, "Many-Core vs. Many-Thread Machines: Stay Away From the Valley," *IEEE Computer Architecture Letters*, vol. 8, no. 01, Jan. 2009.

[4] T. Nowatzki, M. Sartin-Tarm, L. De Carli, K. Sankaralingam, C. Estan, and B. Robatmili, "A general constraint-centric scheduling framework for spatial architectures," in *PLDI*, 2013.

[5] V. Srinivasan, D. Brooks, M. Gschwind, P. Bose, V. Zyuban, P. Strenski, and P. Emma, "Optimizing Pipelines for Power and Performance," in *MICRO*, 2002.