POD: A Parallel-On-Die Architecture

Abstract

As power constraints, microarchitectural complexity, and design verification costs have made it increasingly difficult to improve single-stream performance, parallel computing is beginning to take a place amongst mainstream high-volume architectures. Although there are a few popular approaches in academia and industry that use many-core and/or multi-threading to improve scalar performance, few of these techniques will scale well with future transistor budget increases. As such, most current commercial designs have focused on SMP-style CMPs built by replicating relatively complex single cores using SMP-style protocols and architecture.

While such designs provide a degree of generality, they are not the most efficient way to build chips for applications with inherent scalable parallelism. These SMP-CMPs are designed to be excellent multiprocessors rather than excellent parallel processors. These designs have been proven to work well well for certain classes of applications such as transaction processing, but programming them is extremely complex due to the need to efficiently handle threading, processes, communication, synchronization, and caching effects. These issues are sufficiently burdensome that they have driven the development of new languages and complex architectural features (e.g. transactional memory.) These new approaches can be effective across a variety of workloads, but for inherently scalable workloads, they present substantial language learning-curves and unnecessary hardware features.

Instead of building SMP-style CMPs for all workloads, we propose an alternative parallel on-die (POD) many-core architecture based on a large SIMD MicroCore array. POD helps to address the key challenges of on-chip communication bandwidth, area limitations, and energy consumed by routers by factoring out features that were necessary for SMP and focusing on architectures that match many scalable workloads. In POD, we evaluate and quantify the advantages of a large SIMD array, but we base its ISA on a commercially relevant CISC architecture and show that it can be as efficient as more specialized array processors based on one-off ISAs. Our proposed single-die POD is capable of best-of-class scalar performance as well as up to 1.5 TFLOPS of single-precision floating-point arithmetic. Our simulation results show that in some application domains such as Option Pricing and FFT computation, our architecture can achieve nearly linear speedup on a large number of SIMD MicroCores versus existing complex single-core architectures. For Dense Matrix-Matrix Multiplcation, POD can sustain the floating point computation throughput at 850 GFLOPS with almost negligible inter-core communication overhead.

1 Introduction

Although many current commercial microprocessors have multiple cores and threads on a single die (e.g., IBM Power5 and Cell processors, Intel's Core 2 Quad, Sun's Niagara, and AMD's upcoming quad-core processor), many difficult problems need to be addressed before these designs can make efficient use of their parallel resources. The open questions of how they will be used, what architectural features they will have, and how they will be programmed remain unclear.

Whatever the end-products become, it is very unlikely that a single architectural approach to parallelism will dominate due to vast differences in the behavior of scalable applications. This can be seen by examining existing architectures for transaction processing, graphics, media, and scientific computing. For example, architectures that are efficient for transaction processing are generally ineffective for scientific computing and vice-a-versa; graphics and media processors offer vast computing capacity with relatively lower power consumption, but are so specialized that they are difficult to use for general computing tasks. Beyond the significant architectural questions, the dramatically different programming styles used for these applications indicates a trend towards using domain-specific languages [4, 7, 22, 34] that are typically derived from mainstream serial languages.

This leads to a discussion of how many-core processors will be used and how they will be programmed to fully realize their computational potential. Although a small set of related programming paradigms has largely sufficed for serial designs (C/C++/Java), we believe that such an approach will not work for parallel machines and that a small family of customized languages will be needed. Similarly, while a single architectural design (out-of-order RISC pipelines) has dominated serial architectures, a single approach will not meet the needs of most parallel applications.

In the graphics industry, specialized languages (e.g. Cg [22] or Brook [7]) and standard APIs shield the programmers from the considerable complexity of the underlying hardware design and also provide freedom for hardware designers to innovate from one generation to the next. One possible reason for the adoption of these systems is not that they are on the cutting edge of programming research, but that they have not strayed too far from established serial programming paradigms. In general, we believe that using domain-tailored languages that do not deviate significantly from existing languages and that are customized for their workloads and architectures will demonstrate acceptable usability, programmability, performance, and portability.

In this paper, we propose an architecture that can efficiently support a substantial subset of scalable parallel applications while minimizing the complexity of programming a many-core system. To achieve this goal, we revisit the designs that lead to the data-parallel SIMD computers of the 70's and 80's [3, 35], but we design for the reality of current industry trends in ISAs and programming languages. Since broad acceptance and software compatibility nearly necessitate Intel 64[1] compatibility, we accept this as a basic starting point for our work.

[1] "Intel 32" was previously known as IA32 or x86. "Intel 64" was previously known as IA32e, EM64T, or x86_64.

It has been suggested [13] that MIMD machines won the architectural wars over SIMD in the 90's because they offer two advantages: first, the flexibility of either running a larger number of independent workloads or a smaller number of parallel workloads; second, they leverage off-the-shelf microprocessor economies of scale. In either case, in an age when an entire SIMD array can be placed on a single processor die, both the economies of scale and the flexibility of running multiple independent jobs become possible.

While a SIMD design is not as general as a MIMD or an SMP-style CMP design, it is still well-matched for a very large proportion of scalable workloads and can be programmed with a relatively simple single-control-thread perspective (vs. multi-process, multi-thread, synchronization-based MIMD alternatives). In addition to programming efficiency, SIMD designs also allow for very efficient use of transistors by factoring out common hardware amongst the processor cores, such as complicated CISC instruction decode logic, branch predictors, TLBs, and instruction caches. Even though certain workloads such as transaction processing are clearly inappropriate for SIMD, its fundamental design allows for extremely efficient scaling of a popular legacy architecture, e.g. Intel 64. Finally, the SIMD design allows for a much less complex, much lower latency switching fabric than is required by shared-memory, cache-coherent CMPs.

Our research goals are to provide best of class *performance per watt* across the space of many-cores, graphics processors, and media accelerators while maintaining ISA compatibility and providing a computing model that is more general than the specialized graphics and media processors. The primary contributions of this work over the prior efforts in SIMD machine research are:

- We propose a new Parallel-On-Die (POD) many-core architecture based on a large SIMD MicroCore array.
- Our SIMD MicroCore array represents a first-class citizen (rather than co-processors) in the system with respect to virtual memory and host core.
- We address the wire-latency problems for lock-step execution and communications.
- We eliminate complex global networks and propose an efficient communication topology to address the on-die wiring limitations in our architecture.
- We reduce the need of using thousands of SIMD cores to dozens while attaining multi-TFLOP performance.
- Our architecture maintains semantic compatibility with an existing CISC architecture.

The rest of this paper is organized as follows. In Section 2, we discuss the salient background and critical milestones. Section 3 further explores the contemporary challenges with respect to large-scale SIMD architectures. In Section 4, we propose a modification of a current Intel 64-based microprocessor platform, and in Section 5, we describe ISA support and programming model for POD architecture. Section 6 explains how we constructed our simulator, and Section 7 analyzes the performance results using several commercial benchmark programs. Section 8 concludes.

2 Background

Our work clearly revisits the SIMD concepts that became widely popular in the 1970's and 1980's and expands and interprets them with modern requirements and technological constraints. The closest works (architecturally) to our design are the Maspar MP-1 [3, 23] and the Thinking Machines CM-2 [35]. Due to the close relationship between our architecture and these predecessors, we discuss these machines in depth versus a broader discussion of various SIMD architectures.

Both of these machines were centered on the concept of a very large number of tiles (micro-cores, or processing elements, or PEs) to perform parallel calculations, with the MP-1 having as many as 2^{14} 4-bit processors and the CM-2 having as many as 2^{16} 1-bit processors (plus up to 2^{12} Wietek 32-bit FP co-processors). This style of SIMD machine was broadly characterized by having a front-end system that consists of a host processor or workstation that broadcasts instructions to the PEs, while the host also executes serial code, runs the operating system, and controls multi-programming use of the PE array. Each PE has a small locally accessible memory and a set of computational resources on which its instructions act.

Both machines took advantage of the relatively "free" wire latency compared to the transistor switching speed, and had a low communication latency between nearest neighbors (i.e. 4 cycles for CM-2 and 8 cycles for Maspar). These nearest-neighbor connections were supplemented with more sophisticated networks to allow all-to-all, unstructured, and long-distance communication. The immense number of PEs in either machine mandated that the global network was elaborately designed with many layers of switches and crossbars.

In the case of the CM-2, it provided an n-way hypercube interconnect that linked clusters of PEs to other clusters. The MP-1 utilized a multi-stage, multi-pass network that enabled arbitrary communications patterns, albeit with substantially higher latency and setup costs. Other important SIMD machines included the IBM GF-11 [18], and the Illiac IV [6]. Some hybrid efforts between MIMD and SIMD were undertaken [31], but remain on the fringe. While vector computers [29] are considered to be the most commercially successful large-scale SIMD-style designs, their architecture is so far removed from ours, that we mention them for completeness only.

Systolic arrays [5, 20] resemble SIMD machines, but were tailored for applications whose computations fit a more-narrow structure. In most cases, to be highly efficient, the programmer had to map and understand the application's execution and data flow in great detail to the target machine. The iWarp project [5] is a somewhat generalized representative of this type of machines, but still had a limited application domain and instruction set.

Common SIMD use in modern microprocessors is most evident in the SSE and 3DNow! extensions, as well as the AltiVec or VMX Power Architecture extensions. These instructions were added to baseline architectures to enable programmers and compilers to take advantage of a limited amount of data parallelism for moderate performance improvements. The typical consumers of these extensions are multimedia and game applications.

Imagine [2, 17, 28] is a stream-model processor, but uses a normal host and acts as a coprocessor as we propose.

Imagine uses a 128KB stream register file to contain data, while each attached FP unit has a local register file and several dedicated ALUs to operate on the stream data in a true producer-consumer model, unlike our architecture which uses private SRAMs to allow local reuse of data. Imagine also uses instruction memories and fetch/decode patterns, but capitalized on an 8-wide SIMD ability inside each full ALU tile. The drawback to this design is that each of the eight sub-ALU blocks is wired with a crossbar to the full SRF, and that applications must be ported to a stream-based model for their parallelism to be exposed.

More recently, contemporary tile-based architectures which are not SIMD include the MIT RAW microprocessor and the UT-Austin TRIPS architecture. The MIT RAW processor [33, 32] provides local instruction and data caches, contains 64KB of RAM for each processor tile to program a dynamic/static routers, and has its ALU bypass network tied directly into the interconnect network (ICN). To support its programming model, the RAW project also has large memories and additional modes dedicated to the routing logic for use based on application needs for either static or dynamic routing. While robust from a design perspective, this approach requires the extra logic and power compared to a SIMD design. The TRIPS architecture [30] is tile-based, but uses more sophisticated ISA mechanisms such as predication to allow for compiler analysis and static placement of blocks of computation. It also provides separate ICNs for data and instruction movement and each tile has a complete CPU. The entire design is intended to support highly-speculative parallelization techniques.

The IBM Cell processor [14] provides an alternative approach to tile-based designs by hosting eight specialized processors called Synergistic Processor Elements (SPEs) on a full Power Architecture host. While IBM's Cell processor [14] is similar in concept, these SPEs, MIMD in pattern, are complete with instruction fetch, decode, branch control, and load-store queues. Setting aside the complexity of MIMD programming when compared to the SIMD model we explore, the peak performance of the Cell is below what our architecture can attain. We will make a more detailed performance comparison in Section 4.7.

3 Modern Considerations

One of the primary reasons attributed to the commercial failure of SIMD machines like the Maspar MP-1 and Thinking Machines CM-2 hinges on the rate of growth of microprocessor performance relative to the time to market for SIMD machines [10, 25]. With a concept-to-market time of 36+ months, a new SIMD machine would be released with a scalar performance pegged to the state of the art 1.5-3 years prior to the first sales. Meanwhile, commercial microprocessors leapt ahead by up to a 4x performance improvement following along Moore's Law. Considering the cost of early SIMD machines versus microprocessors, most consumers who could have benefited from parallelization chose not to, letting Moore's law carry on their evolutionary growth as opposed to accepting a major architectural change.

Today, while single stream performance has not reached its limit, it's progress has slowed dramatically due to performance per watt and complexity-effective design [24] issues. While process technology continues to advance, industry leaders look to many-core architectures as the roadmap of the future. With the resultant slowdown in single core improvements, we call in to question the reasons for SIMD machine failure of the past and explore how the original ideas might fit into today's changing landscape.

Our work is based on a very different set of constraints than those that existed when the original SIMD machines were built. First, since we wish to provide an Intel 64-compatible processor and we do not wish to dramatically change its ISA, we cannot overly simplify the processing elements (or called MicroCores in our new architecture). Second, both the Maspar and Thinking Machines designs were not limited to planar interconnects because their processing elements were connected across multiple boards via backplanes and wires. When the CM-2 and MP-1 were built, wire delays were less critical compared to transistor switching speeds. This reduced sensitivity to wire latency is even more pronounced the farther back one goes in the SIMD genealogical tree.

Since we propose to place an entire implementation of a host processor core and a large SIMD array on a single die, we are limited to planar networks [8, 19] and thus do not have the luxury of high-dimensionality networks. This is a key difference in technology over the past two decades. In our design, the wire delay to cross a single MicroCore in a straight line is a function of the manufacturing process, and is intended to be very fast (1 - 3 cycles). In addition to having to adopt to the infeasibility of a global data network, we are also faced with the problem of synchronously broadcasting the SIMD instruction stream across the array of MicroCores in a power-efficient manner. At all stages of design and consideration, wire delays dominated our thinking and drove simplification of the MicroCores. For our initial design, we have restricted the MicroCore size so that a signal can propagate across it in a single cycle.

To simplify the design and minimize complexity, we borrow a similar concept from the interconnect of the MIT RAW processor. The interconnect was directly wired to the pipeline of each RAW processor tile, supporting only nearest-neighbor communications and using fixed algorithms to implement more complex routing. However, because our switches only support single-hop routing and every switch is always communicating in the same direction, the design/size of our routers is much smaller and simpler and provides communications limited only by wire delay. We require no substantial buffering or extra support to handle deadlock, livelock, or drain requirements. We do not use the RAW model of multiple ICN modules within a tile to handle alternatives of static or dynamic routing, and instead use simple algorithms to effect any non-nearest-neighbor communications.

In addition to the interconnect simplification, our design uses far fewer cores than the thousands of cores supported by the MP-1 and CM-2. Since each processing element was only a 1- or 4-bit ALU internally, any given arithmetic operation (e.g., 64-bit integer add) required the use of several processor elements and/or multiple cycles to compute its. Since process technology allows full 128-bit SSE units to be constructed in a small area, this allows us to provide semantic compatibility with the host processor and to reduce the number of SIMD processing elements needed to achieve high rates of computation. In general, our approach allows us to efficiently support

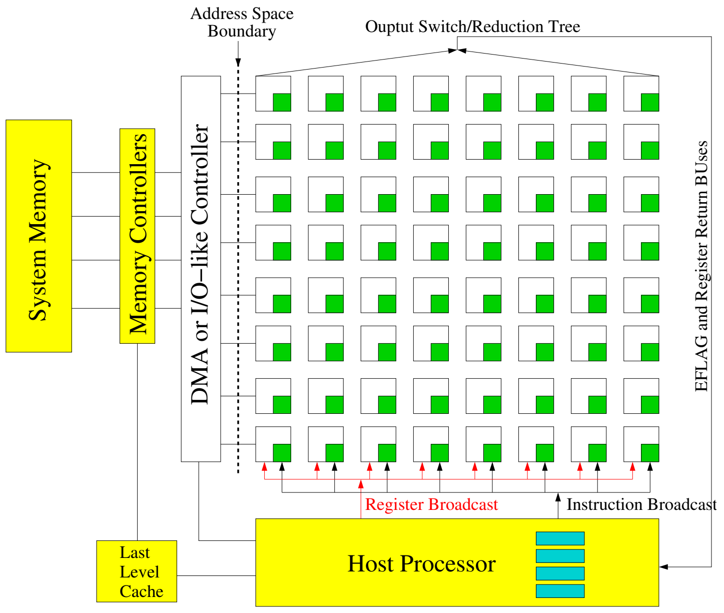


Figure 1: POD architecture with a full host processor core, last level cache, and array of SIMD MicroCores.

most arithmetic operations in the processing elements that are implemented in the host (directly or via dynamic binary translation).

Finally, the SIMD architecture we present is a first-class citizen with respect to the rest of the system. The SIMD array we construct and demonstrate directly interfaces the rest of the system through standard virtual memory interfaces so each MicroCore in the SIMD array can directly access main memory. This is in addition to the private SRAM each MicroCore in our architecture has for local data and computation results.

4 Parallel-On-Die Architecture

In this section, we propose a new massively parallel architecture called *Parallel-On-Die* or *POD*. POD is a fully integrated processing fabric on a single die based on the Intel 64 ISA and provides best-of-class single-stream performance for scalar applications as well as a robust parallel SIMD MicroCore array for scalable parallel application execution. The high-level block diagram of the POD architecture is illustrated in Figure 1. The POD system will fully boot a normal OS and run every legacy application under that OS without problem, thereby presenting the SIMD MicroCore array we attach to it as a pseudo-coprocessor.

4.1 Host Processor Core

The principal claim to best-in-class scalar performance is based on the host core being taken from highest performance Intel Architecture part. A current example could be one core of the Intel Core 2 Duo processor. This provides not only flawless single application execution, but also presents a known, compatible platform to operating systems, programmers, and applications to reduce the complexity of bootstrapping new functionality and applications.

The last-level cache (LLC) shown in this Figure is directly coupled to the memory controllers, which in turn connect to system memory. We illustrate integrated memory controllers, each of which is capable of supporting bandwidth to the main memory dependent upon the technology used (FB-DIMM, DDR2, GDDR3, etc.).

The target SIMD MicroCore array is a sea of $n \times n$ tiles, where we show $n = 8$. The host processor core is capable of broadcasting instructions, each of which has the same fixed size, as well as broadcasting 64-bit register values as might be needed for immediates or loop conditions. The MicroCore array generates a flag-tree output which is tied together logically via *OR* gates, and routed back to the host core.

To allow the addition of a SIMD array while minimally altering the existing Intel 64 ISA, we propose to add a new instruction prefix byte on existing opcodes to denote a parallel-instruction. When this prefix byte is encountered, the host core could implement the instruction by one of several methods. The simplest method is to broadcast it blindly to the POD, but this would require that each MicroCore be able to decode the CISC ISA. A more flexible mechanism is to run a dynamic binary translator or JIT to capture such instructions and selectively decode, broadcast, and optimize them.

For our initial investigations and to avoid the complexity of supporting a new parallel prefix, we instead chose to implement a handful of new instructions that allow us to send native MicroCore instructions from the host. The details of the instruction extension are described in Section 5.1.

4.2 SIMD MicroCore

Each MicroCore tile (or μCore) consists of a high performance arithmetic unit with its own private registers and local SRAM memory space. To provide a baseline performance and power level and to support a subset of the IA instruct set, we chose to assume that we could modify an existing 128-bit SSE engine (including SSE, SSE2 and SSE3) from a contemporary Intel processor. This approach provides 4-wide SIMD execution units for single-precision IEEE floating point operations, or 2-wide for double-precision. We also took the liberty of adding a fused multiply-add instruction to the SSE instructions to improve efficiency of the μCore's resources. By assuming an existing SSE engine design (with extension), we only need to add the surrounding logic to complete a standalone μCore and it minimizes the difference between the host ISA and the μCore ISA. The μCore microarchitecture is shown in Figure 2.

As the Figure illustrates, each μCore has two groups of registers including a 32-entry 64-bit general-purpose register file (`r0 - r31`) and a 32-entry 128-bit SSE-style register file (`xmm0 - xmm31`). While this exceeds the size of Intel Architecture register files, additional resources may or may not be exposed to a programmer directly. In the future, a dynamic binary translator could optimize the host processor's use of 16 XMM registers to make

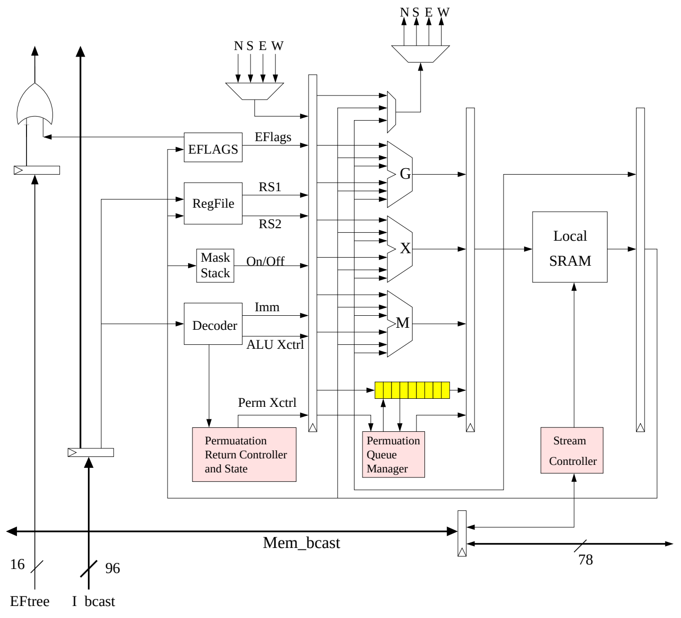


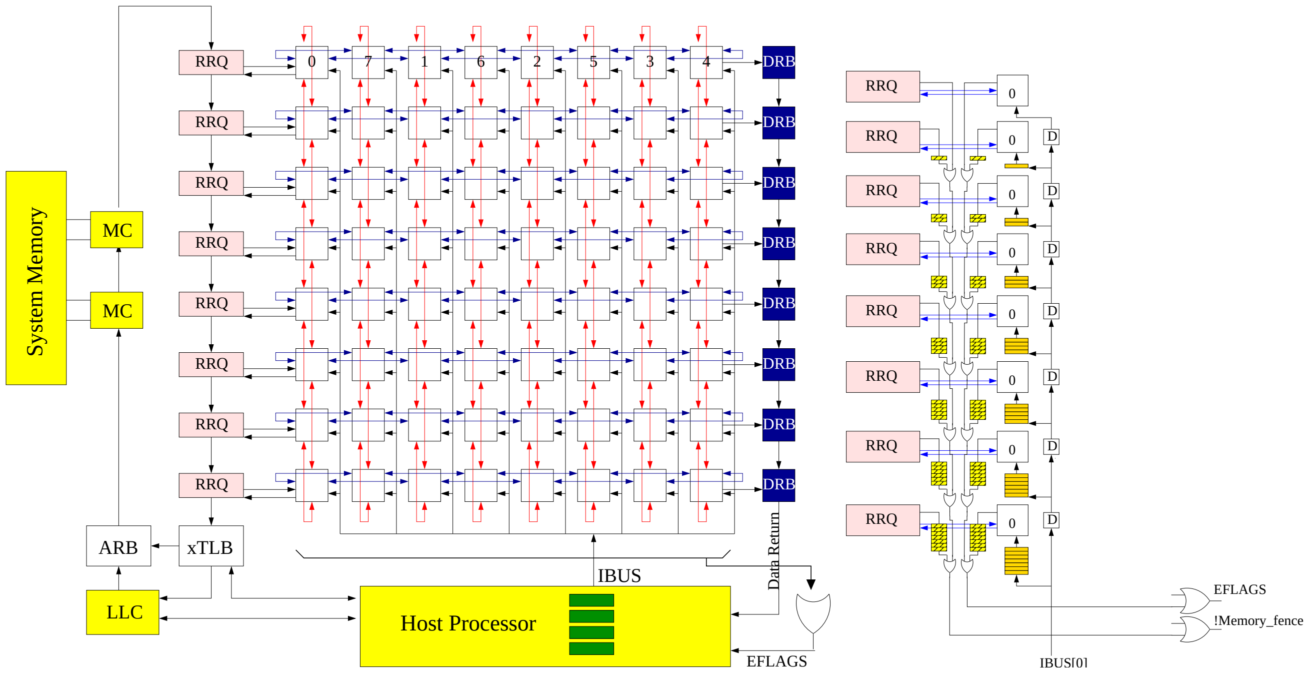
Figure 2: A MicroCore Tile

use of the μCores 32 XMM registers. For purposes of our initial work, all μCore registers are exposed in the POD ISA during hand-coded assembly optimizations.

On the input side of the Figure, the μCore also contains a *Mask Stack*, which is to be used for conditional execution such as *if-then-else* clauses. Our μCore implements a novel way to efficiently execute nested if-then-else clauses, examples of which are in Section 5. On the execution path, there are three individual pipelines within each μCore tile — one for memory and communications instructions (load, store, xfer, etc.) called *M-pipeline*, one *G-pipeline* for generic integer non-SSE arithmetic (address calculation, constant generation, mask operations), and the *X-pipeline* is a full SSE engine supporting all SSE instructions. Based on a 5 to 7 cycle latency for basic integer and floating point operations in the SSE pipeline and 3 cycles to local memory, we require between 15-35 registers to keep each μCore fully running. Our selection of 32 xmm registers satisfies the majority of this range, and requires only one extra bit per source-destination register field in the μCore instruction.

Also shown on the top of the Figure, each SIMD MicroCore consists of four unidirectional input buses from the North, South, East, and West neighbors and four unidirectional output buses from the same neighbors. Each bus is 144 bits wide, capable of latching up to 128 bits of data or register value every clock cycle. These eight buses comprise the data torus for the POD communication patterns. To communicate with main memory, each μCore is further enhanced with two unidirectional memory buses, discussed in Section 4.5.

The μCore instructions are broadcast from the host via a special instruction with an immediate data field of 12 bytes. These 12 bytes forms a partially pre-decoded VLIW packet of three instructions (4 bytes per instruction) that eliminate CISC decoding overhead. Each VLIW packet consists of one G, one M, and one X pipeline instruction. Since the execution of μCore instructions are broadcast and orchestrated by the host, there is no need for an



(a) POD Communication Links

(b) One Detailed POD Column

Figure 3: POD Architecture

instruction cache or associated blocks. Furthermore, since each μCore is executing the same instruction and there is no instruction equivalent to a branch, no branch predictor or associated flush/control logic is required, keeping the μCore small and simple. The salient features of the μCore instruction set will be discussed in Section 5.

The needed control logic is made up of processing arriving μCore instructions, register file and state access, the nearest-neighbor North-South-East-West interconnects via muxes, and a private SRAM accesses. Additional modules are included for the permutation routing control logic to implement complex routing patterns between μCores.

4.3 POD Interconnection Network

In the design of the interconnection network, we investigated two topologies — a 2D mesh and a full torus. We found that while a 2D mesh connect neighboring μCores is straightforward, it has certain drawbacks related to our SIMD routing control. Specifically, when an application needs to communicate from edge to edge, it will take $n-1$ hops. Many parallel algorithms naturally rotate data across μCores (see results from our Cannon's algorithm in Section 7.4). This is a specific problem related to our simplification of the communication network: because all the switches route in the same direction at the same time, and because there is no dynamic routing, the extra flexibility of the torus was required.

As shown in Figure 3(a), our proposed design adopts a modified torus network. To minimize latency and maximize packing, each μCore is designed to take less than one clock cycle for a signal to cross the entire μCore itself. Ideally, each μCore will be no larger in any direction than 95% of the wire distance in one clock cycle with all surrounding line drivers, buffers, and so forth. The ordering of the number labels inside the μCores of the top row in Figure 3(a) indicates the nearest-neighbor connection pattern. In the same way, we lay out the communication links for each column in the POD (north-south direction). In addition to providing shorter links, such a layout also leads to a deterministic communication latency.

As mentioned earlier, there are eight physical ICN buses connected to each μCore. At any given moment, only one direction (input and output) can be is enabled. Since each nearest-neighbor communication pattern has a known latency, the buses are not enabled during periods of pure computation or during periods when buses in the other direction are not being used. This reduced power profile allows growth of the POD array to be limited only by the average power consumption of each μCore and the manufacturing die reticle. This approach is compared to other tiled designs such as MIT RAW or TRIPS or where any of the ICN links could be active at the same time due to dynamic routing.

To enable SIMD-style instruction execution where every μCore executes each instruction at the same global clock cycle, there are two options: (1) execute an instruction immediately upon arrival to a POD row, leading to a North-South *timezone effect*, or (2) buffering each arriving instruction for sufficient time such that every μCore will execute the same instruction at the same instant.

The timezone effect can be challenging to work around for programmers and architects, as any given row will be executing instruction j, while the preceding row is executing $j + 1$ and the successor row is executing $j - 1$. To avoid undesired complexity for programmers, architects, and compilers, we use a buffering model to enable lock-step execution without suffering from the timezone effect.

Figure 3(b) shows such a model for one single column in the POD. Instructions are broadcast using the IBUS and are queued before being executed by the μCore. As shown, the queue size shrinks monotonically as the location of a μCore gets farther away from the host processor. For an $n \times n$ POD, where $n = 8$, there are 7 entries for the bottom-most core while no queue is needed for the top-most core. The delay units (D block) are inserted to delay each instruction broadcast in order to synchronize the SIMD execution. Similarly, when gathering results (*e.g.,* EFLAGS) from μCores, the results from the cores closer to the host processor need to be delayed and wait in their queue till the farther results arrive for combining. These are depicted in the propagation paths with correct delay queues on the left-side of Figure 3(b).

4.4 Interaction Among MicroCores

As mentioned in Section 4.2, each μCore has 4 uni-directional input buses and 4 uni-directional output buses. Each bus pair implements a nearest-neighbor direct link. These eight buses are also arranged such that they are glue-less drop-in components, with each neighboring μCore only requiring bus wiring to complete the layout. This allows for dense packing, although there is a very high wire count.

Each μCore can communicate with its nearest neighbor by either directly moving a register value of up to 128 bits, or by transferring memory in 64-bit chunks. In order to support streaming memory behavior between μCores, we support both single load-store style transfers as well as block-based transfers, with and without striding. Since the nearest neighbor latency for an interleaved torus is targeted to be two cycles or less, this allows for high throughput computation even when the algorithm requires neighboring registers and memory values. This is a major contrast to typical shared-cache interface implementations, where it can take 10 or more cycles to move a value between cores.

When one μCore needs to communicate to another μCore in a non-nearest-neighbor fashion as in k-permutation routing [12], our interconnect design becomes a bottleneck. Rather than provide dynamic wormhole routing hardware support for a relatively infrequent operation, we propose dedicated algorithms to drive the collective POD muxes into a series of sweeps to migrate all data to the intended targets. These algorithms require each μCore to support n hardware buffer slots of the bit-size matching the ICN interconnect bus width.

The basic algorithm proceeds by all μCores send messages to the East, with each message stopping when it reaches its target column. This takes $n-1$ cycles and at the end, at most n messages will be buffered in any one μCore. At the end of this sweep, every message in every POD row will be in its target column. If we now apply the same algorithm to the North, we may require as many as n^2-1 cycles until all the buffered messages reach their target μCore. As messages reach their target μCore, they are processed (stored into the appropriate memory location). This two-phase sweeping algorithm ensures that for *any* permutation of routing, even all-to-one, all messages are delivered after a fixed latency.

While the fixed latency may be high for such generic routing support, we have made the trade-off to keep nearest-neighbor communications fast. More optimized row-only and column-only sweeps of just $(n-1)$ steps are possible for more structured communication to reduce the high latency of a full any-to-any communication.

4.5 Interaction between POD and System Memory

Aside from a 128KB private local SRAM dedicated to each μCore, applications must also be able to communicate with the system through normal loads and stores. To manage this interaction, each μCore is further enhanced with two unidirectional buses to the main memory via an interface called the Row Response Queue (RRQ). One bus streams data back from main memory to the μCores in the row, while the other bus streams data from the μCores

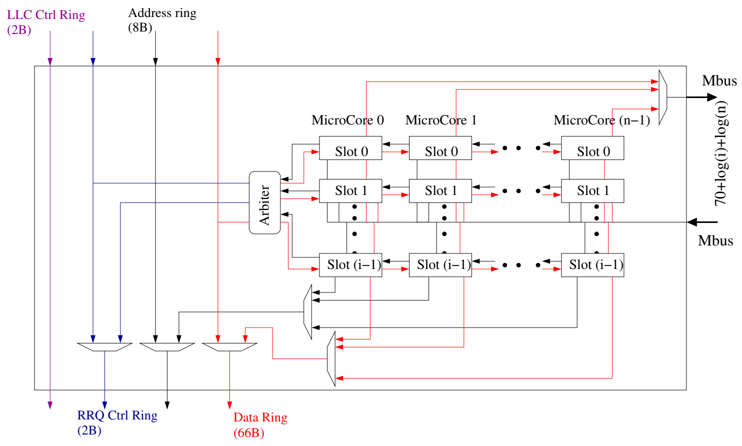


Figure 4: The RRQ state machine for queuing μCore system memory requests and streaming responses from the MCs.

in the row to the main memory. The RRQ is the queuing point for transactions in both directions, and in turn is connected to a memory ring with the IA host core LLC and all memory controllers (MCs).

In our work, the conceptualized ring is composed of four separate rings as shown in Figure 4. There is one shared data ring, at 66 bytes wide, which represents the actual data to or from the MC one line at a time. There is a corresponding address ring of eight bytes to indicate what address a request or response payload corresponds to. Then there are two control rings, one for the LLC and one for all RRQs to share. The premise is that the POD will always be a first-class participant in the memory hierarchy, but a second-class participant to the host core cache misses. Since every request from an RRQ must be acknowledged, whether positively or negatively, when the LLC needs to take over the data and/or address ring for higher priority traffic, an arbiter will set the necessary bits in the RRQ ring for failure and allow the original message to return to the originator for a later retry effort.

One μCore in a row can generate up to i requests in the form of load-store traffic to system memory. Therefore, the RRQ must buffer each request from each μCore and service them as it finds free slots on the ring. Each μCore will only be able to use the i buffers reserved for it, since in traditional SIMD execution every μCore will generate the same number of memory access requests at the same moment, varied only by masking controls.

There are separate instructions for loading and storing to local μCore memory and for the global system memory. Different instruction flavors are provided to load/store single words and contiguous or strided block moves. System memory access use virtual addresses acquired from the host. In order to translate the given virtual address among all n^2 μCores, we share one pipelined TLB external to the host that the host manages. This $xTLB$ in Figure 3 need not be organized along traditional lines since the TLB lookup is not as critical as it is in the host processor – this allows for a super-pipelined, very high capacity xTLB to be implemented. In the event of a fault or miss event in the TLB, the host is notified and the request in the RRQ control ring is flagged as a TLB failure. When the host updates any TLB entry, a dedicated control signal in the RRQ control ring is set to indicate any prior TLB failure may now retry.

Typically, some form of coherence is essential between the collective POD μCore SRAM storage regions and the system memory. To simplify the complexity of our initial research work and to evaluate the basic performance potential of the proposed architecture, we avoid coherence problems by requiring that any memory region that may be loaded into the private POD collective SRAM space to be marked as uncacheable to the host and associated cache hierarchy. While this leads to lower performance sharing of host and POD system memory, it provides sufficient simplification for our initial models.

4.6 Challenges in Integrating with an Out-of-Order Host Processor

Unlike conventional massive SIMD machines, our POD architecture integrates a massive SIMD μCore array with a modern out-of-order host processor. To ensure the execution correctness, there are two major challenges to be addressed in the host processor: recovery from mis-speculation and out-of-order dispatch of POD instructions.

To support speculative execution, some recovery mechanism is required to roll the machine back to the correct architectural state. Unfortunately, implementing recovery mechanism in each μCore will add a substantial overhead to both the area and power. To not complicate the μCore design, we enforce the host processor to broadcast POD instructions in a non-speculative manner. In other words, the POD instructions will not be dispatched from the host until its corresponding branches are resolved. From performance standpoint, as long as the code that runs on the host does not depend on the results from the POD array, this approach will not degrade the performance.[2] Another issue is that the POD instructions might be re-ordered by the host processor. This will lead to correctness problem, because μCore is ignorant of program order. To prevent this, POD instructions issued by the host are strongly ordered, similar to store instructions that are not re-ordered in most of the out-of-order implementations.

To address these issues, we propose an *IBits queue* which is inherently similar to the store queue in a out-of-order processor. When a SendBits instruction is issued, its 12-byte immediate field (encoding a VLIW POD instruction) is entered into the IBits queue. Upon the retirement of the SendBits instruction from the ROB, the corresponding 12-byte immediate value is latched onto the IBus and broadcast to the POD.

4.7 Physical Design Objectives

In our implementation, we aim for a moderately aggressive 3GHz clock speed assuming a next-generation 45nm or better process. For this target frequency, the memory ring is capable of up to 192GB/s bandwidth (servicing up to eight 24GB/s MCs before any modification is required). For an 8×8 POD array, with each μCore containing 128KB of SRAM, connected in a torus, the peak performance of single-precision and double-precision IEEE format FP operations is 1.5 TFLOPS and 768 GFLOPS, respectively.[3]

[2]Note that this is the case for all of our benchmark programs. The host processor does not issue any data-dependent instruction that reads data updated by the POD immediately.

[3]Here are more analytical comparison between POD and IBM Cell. IBM's Cell has a theoretical SP floating-point capacity of 256 GFLOPS for 8 SPE units at 4GHz in a 90nm process [26]. For a fair comparison with our 8x8 POD, we assume that the SPE SRAM

To estimate the overall die size, we use publicly accessible information (based on 65nm Intel Conroe processor) and published literature [27, 16]. First, we evaluate the size of each μCore. Using Intel's public Conroe pictures and floorplan, the integer, SIMD, and AGU pipeline occupies approximately 1.42, 1.36, and 0.14 mm^2, respectively, with a total of 2.92mm^2 in size. The process scaling factor from 65nm to 45nm is 1.44 under perfect conditions, but we assume the scaling of these logic blocks is imperfect to an error of 50% for making conservative estimates. These same units will amount to approximately 2.1mm^2 in 45nm.

For the area of each μCore's local SRAM, according to Intel's published data [1], each SRAM cell at 45nm is approximately 0.346μm^2. A single-ported 128KB SRAM will be roughly 0.363mm^2. Since our basic 128KB μCore SRAM contains 2 Read/Write ports, one Read port and one Write port, we estimate the entire SRAM to be 1.09mm^2. The areas of the two register files and one 32-entry RRQ with 75 bytes each compared to the local SRAM will be insignificant.

Based on these projections, one single μCore will occupy around 3.2mm^2. In other words, the entire 8x8 SIMD μCore array will amount to 205mm^2. Each RRQ, given the complexities of the various bus wirings and the ring interfaces, we allot an equal area on par with each μCore. The total RRQ space is approximately 25.6mm^2. For the host processor, we simply scale the 36mm^2 of one single core in Conroe with the same scaling and fudge factors for 45nm process, the new core is approximately 25.9mm^2. The 3MB LLC is estimated 20mm^2 using the same 45nm SRAM data aforementioned. Therefore, the entire processor will amount to (205+25.6+25.9+20) = 276.5mm^2 without accounting for on-die integrated memory controllers.

5 ISA Support and Programming Model

5.1 ISA Support for Host Core

The SIMD execution inside the POD is completely managed by the host processor. To enable this, we propose extending the host core with five new instructions and modifying three others. Our new instructions are:

- *SendBits*, to broadcast instructions to the POD;
- *GetFlags*, to obtain the return status;
- *DrainFlags*, which assures that the initial setup of a known state in the flag tree is complete;
- *SendRegister*, to broadcast a host register value to every μCore;
- *GetResult*, to obtain a return buffer value from the POD without using system memory as a go-between.

is halved to 128KB and has a perfect shrink with a 2x feature reduction for a 4x increase in number of SPEs, the maximum theoretical performance at 4GHz jumps to 1024 GFLOPs. However, again for fair comparison, the Cell speed should be reduced to our target 3GHz for a peak performance of 768 GFLOPs using all 8 SPEs. In contrast, we attain twice the performance at 1.5 TFLOPs, all while using a simpler μCore design, clocking model, and programming model.

The three modified host instructions are the various *Fence* operations (Load, Store, and combined) that are extended to monitor the return status of the POD's memory interface system. Every other modification that our system requires is *external* to the host core and LLC.

5.2 ISA Support for POD μCore

The instruction set of the μCore supports typical integer/logical instructions, memory instructions, and SSE instructions. The integer and logical instructions operate on (32) 64-bit general-purpose registers while the SSE engine can address (32) 128-bit xmm registers. There are a variety of memory operations supported in the μCore including regular load/store instructions, strided or contiguous block move instructions, and conditional-move instructions. Several versions of memory instructions are provided to allow data accesses from/to local SRAM, remote SRAM on another μCore, and system memory.

To allow multi-level conditional execution in the μCores (nested if-then-else's and while-loops), we provide two types of masking instructions — pushmask and popmask. Inside each μCore, there is a 64-bit mask register that the mask instruction modify to keeps track of the nested conditional state. The top bit of the mask register indicates the masking (on or off) for the current level of a nested control — this allows a μCore to selectively turn itself on or off during the broadcast of instructions from the host. Conditions are determined by flag values which are generated by separate compare operations and their EFLAG results. By turning on and off μCores, it is possible to make only some of cores execute a certain instruction. When entering a new conditional region, pushmask shifts down all the bits in the mask register for each μCore and sets the top bit of the mask register to the new test condition. Whenever leaving a conditional region, the popmask instruction pops one bit out of the mask register and restores the prior state by shifting up. This provides up to a 63-levels of if-then-else or general conditional clauses. If necessary, the programmer or compiler can push or pop the mask register to the system memory or private SRAM to enable higher levels of nesting.

5.3 Mixed Instruction Stream

An example of the basic programming model for our POD prototype is shown in Table 1. Lacking a comprehensive compiler for this architecture, rely on pseudo-C code, consisting of conventional C code for the host processor and annotated inline POD assembly code for the SIMD μCore array.

Our current POD compiler (implemented with a pre-processing script and runtime library), captures this directive and generates corresponding *SendBits* instructions as described in Section 5.1. The host processor is responsible for decoding normal CISC instructions. Once it detects a *SendBits* instruction, the following 96bits comprising our RISC-style VLIW instruction packet will be forwarded to the unit that is responsible for IA-POD instruction broadcast to the SIMD μCore array. Programmers or compilers are required to explicitly generate the code for the

```
$ASM sub8sx r2 = r2, r2
for (int i=0; i<npeX; i++) {
  $ASM add8sx r2 = r2, r1
  $ASM xfer.e r1 = r1
}
$ASM sub8sx r1 = r1, r1
$ASM add8sx r1 = r1, r2
for (int i=0; i<npeY; i++) {
  $ASM xfer.n r1 = r1
  $ASM add8sx r2 = r2, r1
}
```

Table 1: Code Example for POD (ReduceAdd)

μCore array.

Since the latency of all non-system-memory instructions, inter-core communication, and communication between the host and μCore are all determined statically, this programming model is generally free from unrepeatable behavior, difficult debugging, locking, etc. The only unpredictable communication latency is the latency to or from system memory. When references are made to system memory, the host must issue a barrier instruction (one of the host's modified fence operations) before PEs can access the results.

5.4 Inter-μCore Communication

Communication between μCores are explicitly specified by the programmer or compiler as shown in Table 1. While this requires more up-front algorithmic work than an SMP model, it makes the resulting code much easier to debug. Additionally, since the latency of inter-μCore communications is so low, Amdahl's law effects are much less prevalent than they are in longer-latency cache-based designs.

6 Simulation Framework

A cycle-level POD simulator was developed to carry out our performance study. The simulator can sustain approximately 30 KIPS simulation throughput on a 3.4GHz Intel Xeon workstation. However, when simulating a full 8×8 POD, our effective simulation rate is approximately 2 MIPS on the same workstation due to the ability to share pipeline processing in a SIMD array. The feature of not having a complicated instruction fetch/decode mechanism, as well as the lack of control flow, branch prediction, and cache effects on the POD also enables us to attain such high simulation speed.

Our compiler and simulator are tightly connected — the compiler takes the application code and generates a native Intel binary. The assembly instructions are translated by a script, passed through a dependency checker, and schedules POD VLIW packets with latency aware NOP padding, The end result is that the simulation is driven by the host program running natively on the Xeon workstation and every operation that would generate an interaction with the POD, either through broadcast of instructions/data or reading of status/results, instead generates a call into a C++ simulator library.

This library tracks each μCore, every queue, the MC behavior, every private SRAM, and collects various statistics. It also understands the notions of *warp* such that if the host has been busy executing scalar instructions, the next POD-related event will accelerate any in-flight operations on the POD such that the POD clock and the host clock are in virtual synchronization at any moment the POD is actively being used. Each μCore can issue two load/store operations to main memory as well as two block load/store (with or without striding) to main memory concurrently. These operations are asynchronous with respect to the rest of the POD, so the programmer must insert the necessary load/store/memory fence operations to ensure proper behavior. All the various communication buses and links including main memory bandwidth, ring bandwidth, MBUS bandwidth, point-to-point link for 2D torus, etc., is modeled accurately.

There are some limitations in our simulation framework that require improvement in the future. First, delay due to synchronizing with the host processor was not modeled (time to broadcast instruction from the host to the μCores or the time to get a result back from the POD to the host). However, none of our benchmarks needed these features since they are simple computational kernels. Additionally, we did not model overhead in the host processor (Icache misses, branch mispredictions, TLB misses, etc.). However, for the benchmarks shown in this paper and the data sizes used, these affects are unlikely to significantly hurt our results. For future larger and more complex workloads, this feature needs to be implemented. Second, we did not model the LLC takeover of the MC ring, nor did we model the xTLB faults from address translations. Given these activities are rather infrequent with the workloads and datasets we have used, they should cause relatively small inaccuracies in our current results.

7 Performance Evaluation

To evaluate the effectiveness of the POD architecture, we ported a few industry standard benchmark programs and a communication-intensive microbenchmark. To set a baseline, we ran our experiments on a 3.2GHz Pentium D processor with moderately optimized SSE assembly generated by Intel C Compiler. The comparison for the dense matrix-matrix multiplication benchmark was done slightly differently and is explained below. Unless stated otherwise, we modeled a conservative memory bandwidth (6.5 GBps) and a 100ns DRAM access latency for the POD for a reasonably close comparison with the baseline processor.

7.1 Memory-intensive Benchmark: Down-sampling

Our down-sampling (DS) benchmark is an image processing kernel that performs 2:1 down-sampling over a 2112×2112 image where each pixel is represented with three single precision floating point numbers. Over the course of the benchmark, approximately 7 million 1×7 convolutions are computed. Although the algorithm appears to be computation-intensive, as the computing resources on the POD are increased, this benchmark becomes memory bandwidth limited while the cores try to load their data.

This problem becomes clear when one notices that each image requires reading 68MB of data and writing 34MB of results back. When using the Pentium D memory model, this prevents the overall performance from scaling with the number of μCores due to insufficient memory bandwidth. Although POD results do not scale well with the number of μCores, the results in Figure 5 still show a substantial 4x speedup over a Pentium D.

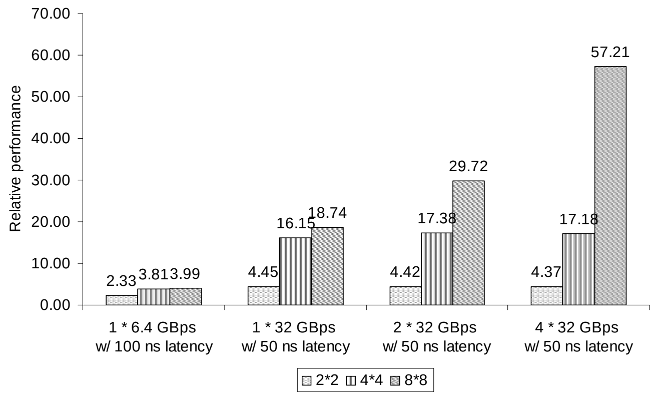


Figure 5: Relative performance of Down-sampling

To get a better understanding on the sensitivity of our design with a more aggressive memory system, we increased the memory bandwidth to 32GBps and lowered the DRAM latency to 50ns. We also scaled the number of on-chip memory controllers from 1 to 2 to 4. Figure 5 shows that in the maximum memory configuration, the POD is able to achieve 57 times the performance of a 3.2GHz Pentium D.

7.2 Computation-intensive Benchmark: Option Pricing

Our option pricing benchmark is a financial application that computes the risk of a portfolio by projecting future options prices (American stock market). The algorithm is computation-intensive and demonstrates high data parallelism. We ported the algorithm based on the binomial tree method [9] to the POD and compared it to a baseline Pentium D version optimized with a moderate effort for SSE instructions.

As mentioned in Section 6, our simulation framework does not model the host at per cycle level. To make the comparisons more reasonable, we measured the execution cycles spent in the same portion of the algorithm on the Pentium D processor and on the POD. As shown in Figure 6, the overall speedup we obtained for an 8×8 μCore array is close to 150 times. The working set for this benchmark is approximately 16KB of data per single-precision element, so a 4-wide SSE engine requires 64KB of private SRAM.

To explain the super-linear speedup, we performed simulations on smaller sized PODs (1×1, 2×2, 4×4, and

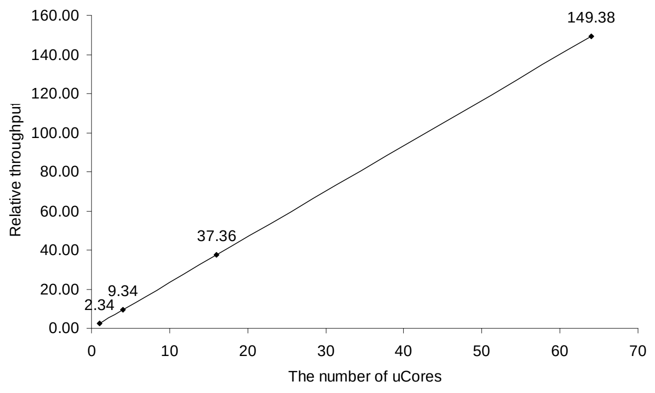


Figure 6: Relative throughput of Option Pricing

8×8). As shown in Figure 6, the throughput of 1×1 μCore configuration is still better than that of the baseline Pentium D processor. The reason for the difference is that we were able to apply modulo scheduling to the inner loop on the SIMD μCores vs the baseline Pentium D SSE implementation. This is not terribly surprising given the very low latency to local memory in the POD, the wide issue μCores, and the increased number of XMM registers. The regular SSE version on the Pentium D was unable to overlap computation with other activities, whereas our VLIW three-pipeline μCore was able to do useful work concurrently. Beyond the single core comparisons, we note that this benchmarks is perfectly data parallel and throughput scales nearly linearly with the number of μCores.

7.3 Communication-intensive Benchmark: Fast Fourier Transform

To test a communication-intensive benchmark, we ported Fast Fourier Transform (FFT) to the POD. We ran a 1024-point 1D FFT on our simulation framework and compared it to a 1024-point FFT on the baseline Pentium D processor using the industry standard FFTW library [11]. On the Pentium D, the FFTW library chooses the fastest implementation for the target machine out of many libraries optimized by using SSE, SSE2, 3DNow!, Altivec, etc.

After running the code initially on our simulator, we discovered that the benchmarks was largely latency limited and that the time to load the initial vectors into the POD dwarfed the computation time. To get a better idea of the speedup potential on small-sized communication intensive codes, we only measured the algorithm after the data was loaded from system memory into μCore local memory. This is not completely unreasonable because many times, high performance applications repeat FFTs several times [15]. Thus we only report the times for the main computation kernel on the POD FFT implementation and compare it with its counterpart from FFTW's optimized library on the Pentium D. The simulation result shows that an 8×8 μCore array with well arranged FFT algorithm can perform a 1024-point 1D FFT **76 times** faster than an SSE-capable processor with FFTW library.[4]

Note that, in each stage of FFT computation, a pair of μCores need to exchange their intermediate results;

[4]Note that part of the speedup could be attributed to the fact that the Pentium D baseline SSE code was automatically generated by Intel C Compiler while the POD FFT code containing more than 1,200 lines of inline-assembly was hand-tuned.

that means, they need to transfer their values in different direction. Transfer to two different directions cannot be done simultaneously in SIMD. To implement this, we split one communication stage into two phases, so that only μCores that are required to store their data at current phase are enabled by using masking operation. Although the FFT requires an arbitrary pattern of communication, that data layout can be modified to behave well on μCore array with $2^n \times 2^n$ μCores. In an 8×8 array configuration, the FFT data set needs to be transferred over 1, 2, or 4 hops depending on the stage of computation, which takes 2, 4, or 8 cycles on the torus respectively. It is worth noting that even the worst-case POD communication latency in this benchmark is better than the L2 access latency of SMP-style CMPs, and furthermore, the latency on the POD can be hidden by pipelined transfers.

7.4 Computation vs. Communication: Dense Matrix-Matrix Multiplication

Dense matrix-matrix multiplication represents the main computation kernel in many linear system problems. We ported Cannon's algorithm [21] to the POD since it is known to be easy to map to a 2D torus interconnect. While matrix-multiplication is obviously computationally intensive, it also requires fast inter-μCore transfers to execute well at smaller problem sizes.

To see how effectively the POD can hide the communication latency, we ran two simulations: 64×64 matrix-matrix multiplication on 1×1 μCore and 512×512 matrix-matrix multiplication on 8×8 μCores. Note that, in the second simulation, each μCore runs also 64×64 matrix-matrix multiplication locally, but requires fast communication with its neighbor μCore. From this simulation, we are comparing the performance of local matrix-matrix multiplication with the performance of global matrix-matrix multiplication to see the communication overhead. The simulation of 64×64 achieved 9.03 GFLOPS while the case of 512×512 achieved 850.83 GFLOPS.[5] The lower throughput of the 64×64 multiplication is due to the overhead of system memory read and writeback and fewer number of computations in the core compared to the case of 512×512 multiplication. To quantify the impact due to inter-core communication, we measured the computation kernel without taking system memory accesses into account. We found that the communication overhead for global computation in Cannon's algorithm is almost negligible.

8 Conclusions

In this paper, we re-evaluate the SIMD computation paradigm in a new many-core architecture called Parallel-On-Die (POD) which integrates a sea of SIMD μCore array into a host processor with minimally new instruction support to enable highly parallel processing. Our POD architecture fills a vital gap between the very general SMP-style CMPs that work well on transactions and multi-programming workloads and the highly specialized processors used for media and graphics-oriented workloads. The SIMD designs also have the advantage that they

[5] The simulation results were based on the configuration with four 32GB/Sec Memory Controllers.

are substantially good fits for implementing highly parallel versions of CISC instruction sets without having to pay the CISC penalty on every processing element. In other words, as one scales a SIMD array to larger sizes, the inefficiencies of the base architecture will be largely hidden, thus making the designs both compatible with existing ISAs and power/performance competitive with more specialized engines.

With the POD-style implementation, it becomes feasible to have both best-of-class scalar performance and extremely efficient scalable parallel performance on a single-die processor with minimally modified instruction set. As shown in our experimental results, nearly linear speedups can be achieved for several commercial applications ported onto our POD architecture. As the industry moves toward the era of 10 billion transistor single-chip processor, the POD architecture will provide a highly scalable and complexity-effective solution.

References

[1] Intel corporation, http://www.intel.com/technology/silicon/new_45nm_silicon.htm.

[2] Jung Ho Ahn, William J. Dally, Brucek Khailany, Ujval Kapasi, and Abhisek Das. Evaluating the Imagine System Architecture. In *Proceedings of the 31st International Sympsoium on Computer Architecture*, 2004.

[3] Tom Blank. The MasPar MP-1 Architecture. In *Proceedings of COMPCON*, Spring 1990.

[4] R. B. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and T. Zhou. Cilk: An efficient multi-threaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, 1996.

[5] Shekhar Borkar, Robert Cohn, George Cox, Sha Gleason, Thomas Gross, H. T. Kung, Monica Lam, Brian Moore, Craig Peterson, John Pieper, Linda Rankin, P. S. Tseng, Jim Sutton, John Urbanski, and Jon Webb. iWarp: An integrated solution to high-speed parallel computing. In *Supercomputing '88*, pages 330–339, 1988.

[6] W. J. Bouknight, Stewart A. Denenberg, David F. McIntyre, J. M. Randall, Amed H. Sameh, and Daniel L. Slotnick. The Illiac IV System. In *Proceedings of IEEE*, April 1972.

[7] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: Stream Computing on Graphics Hardware. In *Proceedings of SIGGRAPH*, 2004.

[8] Andrew A. Chien and Jae H. Kim. Planar-adaptive routing: Low-Cost adaptive networks for multiprocessors. *Journal of the ACM*, 42(1):91–123, 1995.

[9] J.C. Cox, S.A. Ross, and M. Rubinstein. Option Pricing: A Simplified Approach. *Journal of Financial Economics*, 7(3):229–263, 1979.

[10] David E. Culler, Jaswinder Pal Singh, and Anoop Gupta. *Parallel Computer Architecture*. Morgan Kaufmann, USA, 1999.

[11] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. special issue on "Program Generation, Optimization, and Platform Adaptation".

[12] Miltos D. Grammatikakis, D. Frank Hsu, Miro Kraetzl, and Jop F. Sibeyn. Packet routing in fixed-connection networks: A survey. *Journal of Parallel and Distributed Computing*, 54(2):77–132, 1998.

[13] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach (2nd Ed)*. Morgan Kaufmann, San Francisco, California, 2000.

[14] H. Peter Hofstee. Power Efficient Processor Architecture and The Cell Processor. In *Proceedings of the International Symposium on High Performance Computer Architecture*, 2005.

[15] http://www.fftw.org/faq/section3.html.

[16] http://www.sandpile.org.

[17] Ujval J. Kapasi, William J. Dally, Scott Rixner, John D. Owens, and Brucek Khailany. The Imagine Stream Processor. In *Proceedings of the International Conference on Computer Design*, 2002.

[18] M. Kumar, Y. Baransky, and M. Denneau. The GF11 Parallel Computer. *Parallel Computing*, 19(12):1393–1412, 1993.

[19] Rakesh Kumar, Victor Zyuban, and Dean M. Tullsen. Interconnections in Multi-core Architectures: Understanding Mechanisms, Overheads and Scaling. In *Proceedings of the International Symposium on Computer Architecture*, 2005.

[20] H. T. Kung. Why Systolic Architectures. *IEEE Computer*, 15(1):37–46, 1982.

[21] Frank Thomson Leighton. *Introduction to parallel algorithms and architectures : arrays, trees, hypercubes*. Morgan Kaufmann, 1992.

[22] W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard. Cg: A System for Programming Graphics Hardware in C-like Language. In *Proceedings of SIGGRAPH*, 2003.

[23] MasPar. Maspar programming language (ansi c compatible mpl) reference manual.

[24] Subbarao Palacharla, Norman P. Jouppi, and James E. Smith. Complexity-effective superscalar processors. In *ISCA*, pages 206–218, 1997.

[25] Behrooz Parhami. SIMD Machines: Do They Have a Significant Future? In *Proceedings of SIGARCH*, 1995.

[26] D. Pham, S. Asano, M. Bolliger, M. N. Day, H. P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak, M. Suzuoki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki, and K. Yazawa. The Design and Implementation of a First-Generation CELL Processor. In *Proceedings of the 2005 IEEE International Solid-State Circuits Conference*, 2005.

[27] Kiran Puttaswamy and Gabriel H. Loh. Thermal Herding: Microarchitecture Techniques for Controlling HotSpots in High-Performance 3D-Integrated Processors. In *Proceedings of the 13th International Symposium on High Performance Computer Architecture*, 2007.

[28] Scott Rixner, William J. Dally, Ujval Kapasi, Brucek Khailany, Abelardo Lopez-Lagunas, Peter Mattson, and John D. Owens. A Bandwidth-Efficient Architecture for Media Processing. In *Proceedings of the 31st International Sympsoium on Microarchitecture*, 1998.

[29] R. M. Russel. The Cray-1 Computer System. *Communications of the ACM*, 21(1):63–72, 1978.

[30] Karthikeyan Sankaralingam, Ramadass Nagarajan, Haiming Liu, Changkyu Kim, Jaehyuk Huh, Doug Burger, Stephen W. Keckler, and Charles R. Moore. Exploiting ILP, TLP, and DLP with the Polymorphous TRIPS Architecture. In *Proceedings of the 30th International Sympsoium on Computer Architecture*, 2003.

[31] Mikael Taveniku, Anders Ahlander, Magnus Jonsson, and Bertil Svensson. The VEGA Moderately Parallel MIMD, Moderately Parallel SIMD, Architecture for High Performance Array Signal Processing. In *International Parallel Processing Symposium*, 1998.

[32] Michael Taylor, Saman Amarasinghe, and Anant Agarwal. Scalar Operand Networks. In *IEEE Transactions on Parallel and Distributed Systems*, 2005.

[33] Michael Bedford Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffmann, Paul Johnson, Jae-Wook Lee, Walter Lee, Albert Ma, Arvind Saraf, Mark Seneski, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. The Raw Microprocessor: A Computational Fabric for Software Circuits and General Purpose Programs. *IEEE Micro*, March/April, 2002.

[34] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A Language for Streaming Applications. In *Proceedings of CC*, 2002.

[35] Lewis W. Tucker and George G. Robertson. Architecture and Applications of the Connection Machine. In *IEEE Computer*, August 1988.