Chiplets and The Codelet Model

Dawson Fox  
Argonne National Laboratory, Lemont IL  
University of Delaware, Newark DE  
dawsfox@udel.edu

Jose M Monsalve Diaz  
Argonne National Laboratory  
Lemont, IL  
jmonsalvediaz@anl.gov

Xiaoming Li  
University of Delaware  
Newark, DE  
xli@udel.edu

Abstract—The past decade has seen a rapid evolution in hardware technology computation. New architectures that promise to improve performance for domain-specific applications have started to appear. In the future, processors may be composed of a large collection of vendor-independent IP that specializes hardware for application-specific algorithms, resulting in extreme heterogeneity. However, integration of multiple vendors within the same die is difficult. Chiplet, a promising technology, provides a packaging method that integrates multiple vendor dies within the same chip by breaking each piece into an independent block. A common interconnect allows fast data transfer across chiplets.

Most research on chiplets has focused on the interconnect technology. In comparison, programming the chiplet architecture is a less-touched question. Our vision is that it is necessary to define a program execution model (PXM) that enables programmability and performance in chiplet architectures, especially in the handling of multiple vendor-independent chiplets. The key is that a cohesive co-designed PXM can effectively separate the roles of the different actors, while maintaining a common abstraction for program execution. This position paper describes how co-designed Program Execution Models can be used in chiplet architectures. Additionally, it proposes the Codelet PXM as a feasible candidate for a general purpose program execution model in extremely heterogeneous chiplet-based architectures. This paper also presents preliminary examples of how using the Codelet Model can help performance through architectural features and performance models.

Index Terms—chiplets, PXM, performance, portability, programmability

I. INTRODUCTION

For a long time, the computing industry relied on advancements within sequential processors and hardware for better performance. This approach first hit the power wall, which led to the adoption of multi-core shared memory architectures. That architecture is essentially a collection of Von Neumann machines linked by a unified view of memory, each capable of executing independent instructions in parallel and using concepts such as mutexes to synchronize. Unfortunately, the force behind this transition resulted in both extra hardware to resolve issues that arise within parallelism (e.g., enforcing memory consistency models, cache coherency through invalidation mechanisms, atomic memory operations and their related overheads) as well as increased development effort and less predictability on the software side. Now, as Moore’s law slows down, the industry has turned towards heterogeneous architectures that better suit the mix of workloads. This can be seen with the growing prevalence of accelerators such as GPUs, TPUs [18], ML/NN/AI accelerators [12], or specialized sparse computation accelerators. An example of accelerators’ presence in the industry is Apple’s use of the M1 and M2 chips [2] [3], which contain RISC cores, GPU cores, and neural network cores all on the same silicon. Furthermore, cost and power efficiency continue to be paramount in the HPC field due to the necessarily larger scale of the systems involved. Cost and the need for a variety of application-specific architectures creates the need for customized chips; after all, HPC systems have specific needs that cannot all be fulfilled by one vendor.

A solution that has been proposed and appears to hold promise for the near future is the idea of chiplets: each chip containing smaller portions of architectures specialized for a single purpose or field. Chiplets provide a cost-effective way to manufacture smaller, specialized portions of silicon that can be grouped into a single chip, providing the flexibility of integrating different vendors and the heterogeneity to fulfill the specific needs a system might have. However, while the clear benefit of specialized architectures has been proven in a variety of cases, the industry is sorely lacking a cohesive way to organize the execution of codes between highly differing hardware within the same silicon, as well as methods to optimize memory access and data movement within such a system. Recent research in this area has been heavily focused on network-on-chip (NoC) / interconnect and routing algorithms [39] between chiplets or specific chiplet layouts for a particular implementation [30] [32] [37]. While the format of chiplets has previously been explicitly mapped to specific computational problems with due consideration given to software complexity such as in [7], there is a significant gap left between hardware/architecture/interconnect design and software toolchains that are necessary for development. Given that chiplets in a custom chip will likely be developed by different vendors, this gap will widen if a sound program execution model is not selected as a general vision of the system’s architecture.

Meanwhile, parallel architectures are generally left with conventional memory hierarchies. Caches began as an excellent invisible optimization for general sequential program execution and were later adapted to provide correctness and programmability in parallel systems. The sequential past of these designs force parallel programmers to contend with ambiguous memory location and high cache invalidation overheads. Today, taking into consideration the memory wall, some applications that particularly suffer from the built-in protocols of the cache, cache coherency, and prefetching mechanisms use unique memory hierarchies in their corresponding spe-
sional architectures [40] [26] [33]. Techniques such as more intelligent prefetching, smarter memory management based on the execution model, and data streaming through FIFOs can ease this burden more generally.

At a larger scale, the need for load balancing and handling massive amounts of data further drives the need for well-organized execution and memory movement through and between cache hierarchies. This need leads us to the concept of a program execution model (PXM), which already exists in the use of any parallel execution, but may not be explicitly represented. PXMs currently in use do not have clear definitions nor hardware support for their operations, causing them to ultimately fall short of their potential. We view the PXM as the key concept that chiplet architectures need to provide the missing flexibility and programmability. In particular, we propose the use of the Codelet Model PXM, a well-defined fine-grain event-driven PXM, for computation on heterogeneous chiplet-based systems.

This paper aims to convince the reader that, in order to achieve programmability, performance and portability, it is necessary to define a sound program execution model that acts as a contract between users, developers, software infrastructure and hardware designers. In order to achieve this, this manuscript answers the following questions:

- Section II answers the question What problems are likely to be seen when programming on systems using chiplets from various vendors? This section presents a brief example of difficulties that users and developers may experience.
- Section III answers the question What is a Program Execution Model? and Why do we need to define one? This section describes PXMs and their potential effects on performance, efficiency, and programmability.
- Section IV answers the question What is the Codelet PXM? This section describes the Codelet Model, its prior implementations, and its effect as a fine-grain event-driven PXM.
- Section V answers the question Why is the Codelet a suitable model for Chiplets? expressing driving reasons to use the Codelet Model as a PXM on chiplet based systems.
- Finally, Section VI uses a simple Matrix Multiplication case study to demonstrate how the Codelet Model abstraction enables the expression of problems and the mapping of implementation to an abstract machine that may evolve over time.

II. Motivating Example

The problem of not having a well-defined program execution model is inspired by software, but affects all the layers of computer systems. Software has the ability to evolve faster than hardware, and it also allows fast creation and modification of machine abstractions and their behavior. Consequently, current PXMs for parallel computing have been mostly implemented in software, with little direct support in the hardware design.

For programming homogeneous parallel machines (i.e. Multi-core CPUs), the thread abstraction was a natural evolution of software for a von Neumann based multi-core hard-

...
execution. Furthermore, across vendors, there’s not enough information on how these APIs are any different to any other function call, resulting in lack of interoperability. In general, unless a compiler is thought to understand all the possible APIs, it is not possible to obtain semantics of the execution model through API calls.

b) Unified shared memory abstractions: Many runtime systems and chiplet-based architectures are vouching for unified shared memory abstractions, arguing that it makes the code more portable and the system more programmable. However, unified shared memory tampers with the possibility of dataflow analysis and pointer-based optimizations across compilation units. In the example, there’s no guarantee that any of the pointers (A through E) have no aliasing, or that there are hidden data dependencies between multiple API calls (e.g. variables in global memory). Isolation of runtime development across vendors under a unified shared memory, without a cohesive view of the system (i.e. PXM), exacerbates this problem. On the other hand, a pure von Neumann view of the system requires that memory and state are synchronized across different devices. Coherency becomes a fundamental element to support this mechanism but is expensive and difficult to scale. Furthermore, a pure cache-based system misses on performance opportunities that arise from manual memory management. To solve this issue, it is possible to add user managed memory (e.g. scratchpad memory). But it burdens the user once more with memory management, removing some of the programmability gained from a unified shared memory system. While unified shared memory may be a good solution to portability of old code, its adoption is costly and restrictive of more relaxed memory models presented by a PXM.

c) Data, dependencies and parallelism: Firstly, data may require different representations on each chiplet, increasing the overhead of data transformation from one format to the other. Possible additional calls or logic may be needed between each API call in our example to adapt data to the right format (e.g. from column major to row major). Furthermore, in order to properly take advantage of chiplets, parallelism is necessary. The user, or another programming model, is now required to create a new execution model on top of the chiplets API that allows for properly managing execution. Nevertheless, a software only approach is yet another runtime system that needs to be supported to connect all the chiplet’s runtime system. An explicitly defined PXM that serves this role will lead to hardware improvements and reduce the cost of interconnecting different chiplet’s execution models.

To this end, tasking, is a proper abstraction that allows for management of dependencies and parallelism. However, there is currently no tasking PXM that is widely acceptable as a candidate, and, as previously mentioned, a pure implementation in software at the user level will likely generate more collisions with the runtimes used in each chiplet vendor. A generalized PXM can lead to improvements in hardware and Operating System runtimes that free vendors and users from designing these models.

d) Missed opportunities: Finally, optimizations across vendors are often missed due to the lack of agreement between different runtime systems and abstractions. For example, allowing streaming and pipelining across vendors [27] is difficult to achieve in the example code above. It is possible to build software abstractions on top of these APIs, but again it is left to the user or runtime system used by the user. Pointers do not properly expose operation sizes, which does not allow for load balancing. APIs have little information about memory access order, leaving users and compilers without opportunities to improve performance. By using a pure API abstraction, with little information on the execution model used across the program, we are missing on opportunities.

We will now take a deeper look at PXMs and their role in programming and execution.

III. PROGRAM EXECUTION MODELS

A PXM has been originally described as the “formal specification of the application program interface (API) of the computer system” [9]. This brief definition conveys that the PXM is relevant throughout the entire system and that it is necessary to operate it correctly. Dennis et al. [9] also states that a PXM sufficiently describes the system’s semantics for compiler writers as well as providing a goal for designers of a system itself. The PXM in this way constitutes an agreement between hardware and software much like an ISA, but where an ISA is concerned with a single computational unit (a single core, for example), the PXM is system wide. PXMs can be broken down into three models according to [28]: an activity model, a memory model, and a synchronization model. The activity model defines the permitted parallel activities that can occur on the system (in many systems for example, this may be the threading mechanism). The memory model describes how memory is addressed as well as the effects and results caused by operations on the memory. The synchronization model describes the behavior when parallel activities and the memory interact, including the memory consistency model of the system (which on conventional multiprocessors is often realized as Lamport’s sequential consistency [20]). While all parallel execution invokes a PXM of some sort, modern PXMs currently in use are limited.
Much of modern PXMs’ limitations are caused by lack of flexibility or their implementations. PXMs are not necessarily confined individually to hardware or software (see fig. 1). It can be thought of as the API to a specified abstract machine, and therefore the implementations can vary widely between the hardware directly supporting the abstract machine’s model of execution or being implemented solely in software to fit hardware that does not match the abstract machine at all. A modern day example of this is OpenMP, which implements the OpenMP PXM in software. This can be seen in that not all essential synchronization operations/interactions are supported in hardware. Thus, thread creation, tasks, barriers, and synchronizations are not hardware elements but rather software constructs. However, these operations are crucial for correct execution on the OpenMP PXM and yet exist only in software.

On the other hand, when a PXM is implemented or supported in hardware, it still contains components from various levels of the computer system because it represents the overall behavior of the system as a whole when performing parallel computation. True to its definition as a specification of the API of the computer system, it provides procedures that users can invoke, which naturally occurs at the software level. Continuing with OpenMP as the example, one can see how OpenMP clearly provides an API for users to execute parallel programs on a system with correct behavior. The central problem for this section is that, much like OpenMP, modern PXMs are forced to be implemented in software on top of von Neumann abstractions, which inherently limits their performance.

The conventional view of computer systems is too fragmented to facilitate a clear model of parallel computing. The overhead of a PXM is much greater and sometimes prohibitive when implemented in such a fragmented manner. Modern PXMs are built to provide efficient and effective parallelism for user code but forced to rely on interfacing behind the scenes with multiple computational elements that are by and large individual von Neumann machines with a shared memory abstraction. Furthermore, the popular model of parallelism wherein multiple threads have unrestricted behavior and interact in mysterious ways leads to confusion and can make programs near-impossible to debug [21]. Furthermore, data is moved throughout the memory hierarchy and between computational units, largely hampering non-conventional programs and giving little control to user code. The goal of PXMs is the pan-system agreement between all software and all hardware of the system, though they currently fall short of this goal. This agreement should be strict enough to build reliable and understandable programs upon, but flexible enough to allow ease of programming and composability. It should also remain implementation independent (hence, vendor independent), so that innovation in hardware and software do not necessarily have to be dependent on each other. The PXM focuses on interaction of components, so as long as the semantics are enforced, the computational units themselves can be self-contained and need not have any specific execution model within them (note the variety of hardware architectures and runtimes in fig. 1).

In short, a well defined PXM will remove the decidedly unpredictable behavior that so commonly appears in developing and executing parallel applications, and a holistic PXM implementation will support better performance and efficiency. While PXMs currently are lacking as described, the issues will only grow with the arrival of chiplets. PXMs that already result in unpredictable programs will only become worse, as mentioned in section II. When it comes to utilizing multiple architectures with a single program (heterogeneity), the PXMs in use are mostly offloading-based. This model is inherently limiting with regards to achieving high utilization amongst a variety of architectures, most likely involves explicit synchronization on the developer’s part, and can include kernels with unrestricted behavior (possible state change, accesses to global memory, etc.). In a chiplet-based system where these varying architectures are necessarily smaller so as to fit multiple on a single chip, it will be even more difficult to efficiently utilize the components in conjunction with one another. Ideally, a PXM for a chiplet system would include clear definition of behavior, input, and output of workloads. Beyond this, it would provide clear interactions between parallel workloads and memory to achieve more understandable and predictable execution. To this end we propose the Codelet Model PXM, opening the door to better software analysis/optimization, improved isolation, memory efficiency and utilization.

IV. CODELET MODEL

The Codelet Model is a dataflow-inspired PXM to organize computation with an accompanying abstract machine. It is both fine-grain and event driven, breaking computation into Codelets, portions of sequential computation with defined inputs and outputs that are non-preemptive. As such, Codelets are the quantum unit of scheduling of a Codelet-based program. Programs in the Codelet Model are described by a Direct Acyclic Graph, with nodes in the graph representing Codelets and directed arcs representing data dependencies between them. This allows the program to clearly define the ordering of Codelet execution only where necessary while permitting flexibility in Codelet scheduling otherwise. To benefit more from locality, Codelets are grouped into Threaded Procedures (TPs). The Codelet Abstract Machine defines the components of a system that executes Codelet Programs (see fig. 2). It then designates the roles of Compute Unit (CU) and Scheduling Unit (SU): the SU is responsible for creation of TPs and scheduling Codelets during runtime, while the CU is responsible for "firing" (executing) Codelets once their dependencies are fulfilled. SUs and CUs are grouped into "clusters"; the cluster designation is important because TPs can only belong to a single cluster, thus providing the locality mentioned. Codelets, when they have unfulfilled dependencies, are considered dormant. Once their dependencies are all met, they become enabled. When a CU is available to execute the Codelet, the Codelet is considered ready and, having begun execution, is called active. After completing execution, the Codelet can be reset and return to the dormant state, easily facilitating reuse.

For those new to the Codelet Model, a common question is "how are Codelets different from threads?" A very simple answer is that Codelets are lightweight, non-preemptive and stateless. A user may write a thread that yields and later resumes execution for example; by definition, a Codelet may
Codelet present the opportunity for smarter data movement, preemptive. In fact, the well-defined inputs and outputs of each potential latency hiding that is caused by Codelets being non-type of dependency. This concept helps mitigate the loss of implementations. An example of an extended Codelet Model "cluster" may not contain only identical CUs, CUs have been viewed as identical from the perspective of the SU. However, as long as the system throughout supports the mechanisms of signaling between Codelets to track whether or not dependencies are met, and as long as Codelets are written in a way such that they can be correctly executed on a given CU, the Codelet Model will have correct execution. A straightforward example of this would be if multiple CUs have the same ISA but differing memory hierarchies (different sized caches, cache lines, etc.). Writing the Codelets would not necessarily require any changes; however, with added metadata to indicate targeting a certain CU and with either compiler optimizations or an experienced developer, more efficient execution could be achieved. Now we will explore some of the benefits that have been gained from using the Codelet Model in the past.

The Codelet Model has also been used to define a hierarchically structured von Neumann machine. A program for this machine abstraction is envisioned as a hierarchical description of Codelets. Parallelism is achieved through techniques inspired by instruction level parallelism such as out of order execution engines and superscalar architectures. This work also considers heterogeneity, and enables compiler optimizations to take advantage of program execution.

B. Prior Codelet Model Implementations

DARTS (Delaware Adaptive RunTime System) is the basic software-level Codelet Model implementation that is the basis of prior work on improving parallelism and efficiency using the Codelet Model. More specifically, various Codelet Model implementations have already been shown to offer benefit to conventional computations. Take, for example, the use of the Codelet Model to compute fine-grain FFTs. The advantage of the Codelet Model’s ability to provide fine-grain parallelism can help achieve better utilization and load balancing. In addition, DARTS is an example of how the clear memory inputs and outputs of Codelets can be leveraged to optimize memory accesses, balance the application, and further exploit locality. A fine-grain Codelet Model implementation was also used to target parallel stencil computation in. It can be seen there that speedup was achieved over coarser-grain models, but the limitations of a software only PXM can be noted as well: the performance was somewhat dependent on von Neumann cache behavior and platform dependent mechanisms. This once again indicates the need for a cohesive PXM throughout the varying levels of the system to make advancements in performance and efficiency generally which could only otherwise come from platform dependent implementations. An example of an extended Codelet Model implementation can help us demonstrate this.
DARTS has previously been used to include streaming as a method of performing dataflow software pipelining at the Codelet level [29]. The implementation used FIFO queues to perform the pipelining and was tested on GEMM using Cannon's algorithm; it was shown to provide speedup, better computational efficiency, and lower synchronization overhead than the non-pipelining implementation. There is undergoing work to integrate the implementation of [29] into the runtime itself as an effort to increase programmability and generality of the streaming mechanism and apply the mechanism to more complex computational problems. The streaming Codelets could then be assigned the FIFOs by the scheduling mechanism and access them only using typical push and pop methods, effectively providing fine-grain pipelining between them at very low effort to the application developer. This demonstrates well the potential the Codelet Model PXM misses out on without hardware support: on a typical von Neumann multicore machine, without a HW implemented FIFO, it is completely ambiguous to the runtime and to the application developer where the FIFO is physically stored, and it may be affected unpredictably by mechanisms like cache coherency and false sharing.

Unlike the addition of dataflow software pipelining which mostly provides better performance at the cluster level, advancements can also be made by extending the Codelet Model to handle system wide scheduling better. Within the Codelet model, there is a native two level hierarchy of computational units: the Codelet, which is the principal scheduling quantum of the PXMs, and the Threaded Procedure, which groups closely related Codelets to gain locality during their execution. Within the level of a single system, the SUs of the Codelet Abstract Machine (CAM) manage creation of TPis and distribution of Codelets. When looking at larger systems such as an exascale supercomputer, there is a need for a higher level scheduling mechanism as well. Such work has begun with the creation of DECARD [30]. DECARD fulfills the need of a dedicated scheduling mechanism at the system level that can smartly and efficiently distribute TPis throughout a system and manage communications between the nodes. It is implemented primarily through finite state machines that could be implemented at the register transfer level and later included as part of the hardware design. DECARD can be viewed as an extension of the Codelet Model PXMs for better efficiency, load balancing, and performance at a large scale. The multiple examples mentioned in this section lend credibility to the Codelet Model PXM on currently used systems, but we now propose that the Codelet Model could provide even greater benefit on the systems of the future.

V. THE CODELET MODEL AS IT PERTAINS TO CHIPLETS

Already, much exposition has been given on the prior work done and the potential benefits provided by a fine-grain event-driven PXMs like the Codelet Model, even when only implemented as a software runtime. Now, we describe how the Codelet Model can provide a PXMs on chiplet based systems for improved composability, utilization, and memory efficiency. There are many potential PXMs that could be implemented on a chiplet-based system. Based on prior trends, it seems probable that a chiplet-based system will rely on an offloading-based PXM, where kernels or tasks are pushed to "coprocessors" (which in this case are in fact resident on the host processor chip). This introduces many problems; though kernels may appear at a glance to be Codelet-like tasks, they are in reality simply function calls and can have unpredictable behavior like global memory accesses or state changes. This model of computation is similar to a tasking PXMs, but the underlying problem is that the PXMs implemented at the hardware level has absolutely no concept of a "task", which allows such a task to have wide communication and side effects. The magnitude of these effects may only grow in the presence of unified shared memory, because the task can then have direct effects on host memory. Such unified shared memory may even impose von Neumann abstractions on the memory which would then incur the same issues that exist on chip multiprocessors now: coherency issues, consistency limitations, unrestricted behavior, and all the overhead and performance degradation that may accompany them. The Codelet Model can be a solution in various ways.

A. Programmability and Understandability

Firstly, chiplets are a powerful example where the Codelet Model can leverage its definition to achieve programmable heterogeneity with predictable and understandable execution. As long as the behavior meets the standards of the Codelet Model, the only necessary difference would be how the Codelets themselves are written by the developer. However, as the developer would typically have to explicitly designate code to be executed on the non-CPU cores in something akin to an offloading or coprocessor model, such code or kernels are likely well suited to be Codelets (barring the unpredictable behavior mentioned above) and the extra development effort would be minimal. Then, it follows that the Codelet Model can provide heterogeneous parallelism, since any Codelets that have their dependencies fulfilled (any enabled Codelet) can be executed on the respective CU (CPU core, GPU core, etc.). The parallelism inherent in the program is exposed through the structure of the Codelet Graph and any explicit scheduling or synchronization is no longer the burden of the developer. It should be mentioned that in such a case as the existing chiplet systems [37] [36], the implementation of the Codelet Model on the system will be limited (as the systems likely do not have direct hardware support for Codelet Model operations) and that due to differences in system layout and memory hierarchy, optimizations should be unique to the platform on which they are implemented. Overall, the Codelet Model can help maintain programmability at least at a level similar to an offloading-based model. Such is one of the advantages provided by a clearly defined PXMs, but the Codelet Model offers performance benefits as well.

B. Improving Utilization and Memory Efficiency

The Codelet Model PXM can improve utilization issues, even on a system with various chiplets. Because Codelets have well-defined inputs and outputs, it is known at compile time what other Codelets will produce the data needed, as well as what other Codelets will need the data produced. The freedom of being able to organize smaller tasks with clear data dependencies can allow more optimization to occur with
the order of memory accesses and their collisions (even at the software level \[38\]), which will grow more important if each chiplet or at least various chiplets have their own local memory. The benefits here will be implementation dependent and rely on the predictability of memory accesses. For example, a scheduling policy could be implemented that is tailored to avoiding memory bank collisions. Thanks to the graph structure of the program and clearly defined input and output of Codelets, optimizations could be made which are not feasible on less-defined program structures like kernels or tasks.

Such clearly defined input/output of Codelets also opens the door to smart scheduling decisions on the chip. Already, software implementations of the Codelet Model are relatively lightweight and generally do not incur very much synchronization overhead \[34\]. Consider that with many distinct architectures on a chip, it may become difficult to properly distribute tasks in an effective manner. The Codelet Model presents an opportunity to utilize all chiplets on a chip wherever the parallelism inherent in the problem allows it to, since it indicates clear dependencies. This allows the scheduling mechanism to make the most of the program graph and allows the developer to make Codelets as fine-grain as allowed by the algorithm; however, the size of Codelets must be balanced against the scheduling/synchronization overhead in terms of utilization. To let Codelets be finer and expose more parallelism in the system for better utilization, synchronization overhead and scheduling overhead can be reduced with the help of hardware support.

C. Possible Hardware Augmentations

As mentioned, the Codelet Model PXM can never meet its full potential (nor any other PXM) without hardware that supports operations specific to the PXM and its corresponding abstract machine. There are many possible ways to support the Codelet Model in hardware, varying from small differences that have little effect on conventional parallel computation to those that would have a large effect on it. There are already some examples of hardware supporting different PXM that help demonstrate the benefit of adding hardware support. As one such example, Taco \[51\] includes near-cache reusable dataflow execution engines that can be programmed at the high-level through definitions of functions such as onMiss (to trigger on a memory access that registers as a cache miss). What is most interesting here is the clearly exploitable interface which is provided through overloading functions; in Taco’s PXM, operations that occur on cache misses or cache flushes are part of its abstract machine, though this is not clearly stated, and implemented in hardware to facilitate high performance. It is an interesting example of how hardware support for alternative non-conventional PXM can help improve execution of applications \[51\]. It is also a specific example of the benefits that can only be exploited by tailoring the movement of data throughout the memory hierarchy to fit an alternative PXM.

As to how the Codelet Model could be augmented through hardware support, there are multiple options. A simple motivating example is hardware dedicated to tracking Codelet dependencies. In the software based Codelet Model implementation DARTS, a data structure called a Synchronization Slot is created to hold a counter of the Codelet’s dependencies and decrement it each time another is fulfilled \[34\]. This could be implemented in hardware through a counter that has (1) a current dependency count, (2) a reset dependency value (which is employed for initialization and Codelet reset for reuse), (3) sufficient logic to decrement and reset the count, and (4) the ability to receive decrement and reset signals. In DARTS, decrementing a Codelet’s dependency is a necessarily atomic operation, meaning that the general flow of the decrement process involves fetching the dependency count value, decrementing it using the resident ALU, and storing the value back to its location. This process takes at a minimum twice the access latency of the dependency count’s physical location, which is ultimately ambiguous both to DARTS and to the user, all while maintaining exclusive access to the memory location and potentially delaying other signalling further. In the hardware augmented case, each data structure could receive signals from multiple CUs and serialize them with basic arbitration. Furthermore, there would be no coherency issues since Codelets are owned by a TP, and TPs belong only to a single cluster. This simple mechanism would serve to reduce the signaling overhead when compared to the software implementation by reducing the latency of the operation and not requiring exclusive access.

Another possible hardware addition could include hardware Codelet queues. DARTS provides the CUs Codelet queues from which to pop enabled Codelets to execute, while the SU has a Codelet queue from which to distribute them \[34\]. Once more, the physical location of these queues is ambiguous, and with a typical multicore cache hierarchy, the queues or their contents may be unexpectedly evicted from a local cache or invalidated due to operations from other cores (such as pushing a new Codelet to the queue). These invalidations are costly, and may also result in the data needing to be fetched from a higher level of the memory hierarchy repeatedly since the queues are accessed often. Implementing this queue in hardware should be rather straightforward as Codelets do not physically contain their code but simply point to it, resulting in all Codelets being of uniform size and structure. This idea could further be expanded to include a prefetching mechanism for the code itself, though this may be implementation dependent. As the scheduling policy may also be implementation dependent or even reconfigurable, in that case the queue should be sufficiently flexible to provide the operations needed at the hardware level; the queue should be well implemented to support the scheduling policies that are available in the given platform and implementation.

Lastly, as we have already seen, Codelet Model implementations have benefited greatly from including data streaming mechanisms to support pipelining of Codelet execution \[29\]. This could be implemented using FIFO (First In First Out) queues that support general push and pop interfaces. When used similarly to \[29\], access to the queues could bypass cache mechanisms entirely and relieve the burden of the cache coherency mechanism during simultaneous fine-grain accesses to closely-located memory locations. If the push and pop interfaces are blocking, then there is also the opportunity to avoid polling mechanisms which are often present in
blocking push and pop calls to software implemented FIFO queues. This can help increase efficiency and speed while still providing fine-grain synchronization between two or more concurrent Codelets that would otherwise be forced to execute sequentially.

D. The Case Against Cache Coherency

As mentioned previously, much of the prior research done on chiplet-based chips is related to interconnect technology and protocols [39] [6]. As a consequence, the discussion on cache coherency must be mentioned, since cache coherency mechanisms necessitate low latency interconnects to reduce their effect on shared memory multiprocessing and this naturally affects the design of interconnects for chiplets [6]. We propose removing coherency at specific or all levels of the memory hierarchy to lessen the traffic on the interconnects and allow them to dedicate more resources to data transfer instead of inter-cache coherency messaging. While this certainly will seem radical, it is significantly more tractable under the implementation of the Codelet Model with hardware support. In support of this idea, note that since multiple TPs do not share the same data elements concurrently, coherency mechanisms across TPs are not needed at all and could be completely removed if TPs had their own level in the memory hierarchy. The only data that is conceptually shared between TPs is the results of one TP that another TP requires as input; this data transfer could be achieved through a message-passing manner that is implicit as part of the continuation codelet instead of utilizing typical shared memory mechanisms, avoiding the coherency problem altogether. This data transfer between TPs could also safely be achieved through I-Structures [8].

At the Codelet level, no two Codelets that write or overwrite the same data element can execute at the same time by definition; in this way, concurrently executing Codelets are effectively isolated in terms of data accesses (except in the case of pipelining, though this problem is then resolved through hardware FIFO queues). Generally speaking, Codelets should then require no coherency mechanism in any memory structure that is not shared (such as a conventional L1 data cache). In such a case, new write-back policies could be developed based on Codelet Model semantics between Codelet-level memory and TP-level memory. Though this is a much more significant divergence from conventional PXMs in the sense that it has the potential to break most non-Codelet Model programs run on the system, it would be palatable to add configuration of the cache such that it can toggle active coherency mechanisms. Another alternative is that caches on chip could have the possibility to be configured as scratchpad memory much like in the Polarfire SoC chip [1]. This would allow the Codelet Model PXM to leverage it as a fast, local memory that could be used intelligently based on Codelet Model semantics for data reuse, prefetching, and movement. If caches could be reimagined as portions of scratchpad memory that are configured as having frame allocation for each Codelet, mechanisms could arise that facilitate more direct transfer of input and output between Codelets. Using hardware-implemented lightweight queue structures could bypass the cache mechanism and move much of the data transfer or data “writing” to actually passing the new data directly to the Codelet. In other words, shared caches could be replaced by more manipulable counterparts in order to leverage the semantics of the Codelet Model for more efficient memory management at the level of a TP.

Beyond mechanisms that particularly rely on the semantics of the Codelet Model, other ideas can be considered such as the Location Consistency model [14] [22]. The (Enhanced) Location Consistency model has been shown to be weaker than currently used models while still producing results equivalent to sequential consistency in data race free programs (which well-formed Codelet Model programs are).

Specific implementations, protocols, or scheduling policies in support of this idea are out of the scope of the paper and should be extrapolated on in other publications. However, we believe this is an interesting way to improve performance especially considering the memory wall while maintaining composability and programmability.

VI. CODELET MODEL ON CHIPLETS CASE STUDY

In order to give tangible support for the ideas expressed in this paper, a small case study has been developed to demonstrate the use of the Codelet Model in modern problems. A small Python-based Codelet Model simulator that assumes ideal scheduling and abundant bandwidth allows us to explore how the Codelet Model behaves with the inclusion of pipelining and chiplets. While the simulator is high-level and makes some assumptions, it does follow the execution of a an actual Codelet Graph that could be used in existing Codelet Model implementations. After this is further discussed, explanation will be given with respect to the limitations of the simulator.

A. Codelet Model Simulator

The simulator used for this case study can be seen as a sequential, centralized scheduler of Codelets that replaces execution time of Codelets with a defined delay. It keeps track of each Codelet’s dependency counter and modifies them when prior Codelets are completed. The signalling between Codelets incurs no delay and because the scheduling mechanism is ideal, the memory needed for the Codelet is assumed to be already resident to the CU it is executing on. The scheduling mechanism is also assumed to not incur overhead.

The simulator can be run with various options, including the number of CUs and whether or not it is utilizing pipelining/streaming and chiplets. When executing the simulator using chiplets enabled, the number of non-chiplet CUs should be adjusted so that it does not in effect “widen” the hardware that is being simulated. The simulator aims to show the advantage of using chiplets on the Codelet Model, and this would be skewed by the addition of compute resources instead of the replacement of compute resources with ones tailored to specific applications.

Chiplets are implemented as CUs of a specific type that can only execute Codelets enabled for that chiplet type. A “special” Codelet targeting a specific chiplet applies a related speedup multiplier, reducing the time spent active before signalling the next Codelets in the graph. In particular, two chiplets were used in the results shown below: one meant to emulate TPUs [18] in their ability to perform matrix and vector operations very well, and another meant to emulate the
Fig. 3: A 4 tile Codelet graph implementing the outer product method of matrix multiplication that is in a sub-optimal format. The Codelet graph converts the matrix to a desirable format, performs multiplication of one column of matrix A and one column of matrix B (at finest grain), producing a partial result matrix which is summed with all other partial result matrices.

The benefit of accelerators like UDP [13], which speeds up branch-heavy tasks such as format conversion, pattern matching, or compression/decompression. UDP can be embedded as near-memory compute instead of a traditional accelerator, providing benefits towards pipelining. The speed up multipliers applied to the Codelets targeting chiplets in the simulator are 30x and 10x, conservative estimates of TPU [19] and UDP [13] speedup respectively to keep them grounded in reality.

Pipelining similar to [29], on the other hand, is implemented practically as the ability to concurrently execute a Codelet with the Codelet that it depends on. If a Codelet is marked as pipeline-enabled, upon being enabled it will also enable its consumer Codelet so that their execution can be overlapped. Pipelining can occur through an arbitrary number of Codelets.

B. Matrix Multiplication Results

Two algorithms for performing GEMM were tested on the simulator: the inner product method and the outer product method. While less common, the outer product method can present more opportunities for memory reuse, which can be important when memory access is the limiting factor. Both methods were tested at various sizes and with varying numbers of compute elements. The Codelet graphs for these algorithms are relatively straightforward. For example, the inner product method can be realized as a separate Codelet for each tile in the result matrix, each independent from the other (at smallest tiles possible, each tile represents a single element in the result matrix). One important note about this is that the graph for each GEMM method is effectively based on the number of tiles, and for that reason is not explicitly tied to any specific matrix size. It should be noted that the Codelets would be coarser in a realistic situation (larger tiles) so that the program is not overly burdened by synchronization and scheduling overheads. The outer product program graph under test can be seen as a 4 tile example in fig. 3. Associated traces of the simulated execution can be seen in fig. 4 with various configurations.

The same graph has been expanded up to 64 tiles. Note that as the graph grows wider, the inverted-tree-like structure of the matrix sum Codelets also grows taller. For example, the same graph expanded for 8 tiles (Codelets wide) has 7 matrix sum Codelets across 3 "levels" of the graph. The results of the varying number of tiles can be seen in tables I and II. Notice that the version that consistently performs the best is with pipelining and chiplet usage combined, but that it begins scaling poorly in the outer product method in terms of speedup once number of tiles reaches 64. This is because in this example, the two chiplet types are given equal amounts of resources (about 32 each). Once the tile count reaches 32, the best speedup is shown because this is where the amount of each resource matches the number of tiles (and therefore, width of the graph) and the pipelining allows the entire matrix sum portion of the tree to execute concurrently, conceptually streaming the data throughout the CUs and pipelining the entire sum. After the number of tiles grows to 64, the speedup is not as substantial because the resources are saturated while they are not in the non-chiplet cases (since the 64 CUs are all conventional). The speedup can also vary based on the ratio of resources contained by one chiplet to the other chiplet. Beyond this, part of this change in speedup could be due to the way the Codelet delays were modeled: the Codelets representing the format conversion and vector multiplication scale by O(Tiles) while the matrix sum Codelets scale by O(Tiles^2), meaning that the benefit of each chiplet also scales unevenly.

Alternatively, the speedups achieved for the inner product method are less significant than those achieve for the outer product method. This is because the inner product method already exposes as much parallelism as possible based on tile
width, and as a result benefits very little from pipelining. Also, it most likely suffers more severely from the limited number of chiplet CUs dedicated to speeding up the core calculation portion, since this conceptually results in the chiplet version having less CUs than the normal version. Having less CUs available to execute the Codelets also further limits the benefit of pipelining in the simulation since there is no memory behavior being modeled in the simulator (see section VI-C).

Furthermore, the sequential centralized simulation grows increasingly taxing as the tile count increases exponentially. As a result, we are currently limited to 64 tiles. Testing different ratios of chiplet resources or by making matrix sums finer grain is left for future work.

C. Simulator Justification and Future Work

It is clear that our current implementation of a Codelet Model simulator for analyzing the performance of Codelet programs is limited. However, the results and the simulator can be justified as it pertains to a full-system Codelet Model PXM implementation with hardware support. Firstly, critics of the simulator will notice that it does not account for signalling and scheduling overhead. We posit that the overhead incurred by signalling can be mitigated through a hardware mechanism like the one mentioned in section VC. Latency incurred by that operation in most cases would be hidden by the scheduling and execution of other already-enabled Codelets. Similarly, the scheduling overhead could be mitigated through hardware queues also mentioned in section VC. Other scheduling overhead components such as the latency of the scheduling decisions are left for future work.

One thing lacking in the simulator is memory behavior. With a good enough scheduling mechanism, we can assume that Codelets are scheduled where their input data are already resident. Beyond this, we predict that the benefit of pipelining/streaming in this simulator is underestimated. In a realistic system, the execution could benefit from HW FIFOs bypassing caching coherency mechanisms and ensuring low latency access to the data. In the simulator pipelining effectively only increases the number of Codelets that can be enabled at a single moment based on the Codelet Graph which improves utilization. In the future we expect to improve this simulator to include memory behavior and use it as a tool to develop better memory-conscious scheduling decisions.

VI. Conclusion

Chiplets are the hardware answer to industry’s constant push for improvement in multiple problem areas, but will introduce even more difficulties to software development methods that are already fraught with issues. In turn, restrictions and difficulties in software development will limit the efficient use of the hardware itself. The answer is a clearly defined PXM integrated at all levels, and the fine-grain event-driven Codelet Model PXM presents new opportunities to improve performance and efficiency while maintaining understandability and clearly defined interactions. Improvements made to the Codelet Model such as FIFO queues for dataflow software pipelining can be implemented at the hardware level, helping to improve utilization and limit memory access latency within Codelets. In the future, we hope to further show the validity of the Codelet Model as a solution through improving the simulator to include memory behavior and a wider array of applications. While utilization and speedup speak for themselves through simulation, other stated benefits like understandability and composability of programs cannot be quantified, but will be known to all those who have spent sleepless nights debugging a multithreaded program.

REFERENCES

[1] Dec 2021. [Online]. Available: https://onlinedocs.microchip.com/pr/GUID-0E320577-28E6-4365-9BB8-9E1416A0A6E4-en-US-3/index.html
[2] “Apple unleashes m1,” Aug 2022. [Online]. Available: https://www.apple.com/newsroom/2022/06/apple-unveils-m2-with-breakthrough-performance-and-capabilities/ Aug 2022. [Online]. Available: https://www.apple.com/newsroom/2022/06/apple-unveils-m2-with-breakthrough-performance-and-capabilities/
[3] S. Bharadwaj, J. Yin, B. Beckmann, and T. Krishna, “Kite: A family of heterogeneous interposer topologies enabled via accurate interconnect modeling,” in 2020 57th ACM/IEEE Design Automation Conference (DAC), 2020, pp. 1–6.
[4] A. A. Chien, A. Snavely, and M. Gahagan, “10x10: A general-purpose architectural approach to heterogeneity and energy efficiency,” Procedia Computer Science, vol. 4, pp. 1987–1996, 2011, proceedings of the International Conference on Computational Science, ICCS 2011, [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050911002754
[5] D. Beckingsale, R. Hornung, T. Scogland, and A. Vargas, “Performance portable c++ programming with raja,” in Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming, ser. PPoPP ’19. New York, NY, USA: Association for Computing Machinery, 2019, p. 455–456. [Online]. Available: https://doi.org/10.1145/3293883.3302577
[6] S. Culler, “Dataflow architectures,” Annual review of computer science, vol. 1, no. 1, pp. 225–253, 1986.
[7] J. Dennis, “A parallel program execution model supporting modular software construction,” in 3rd Working Conf. on Massively Parallel Programming Models, Nov. 1997, pp. 50–60.
[8] J. M. M. Diza, K. Harms, R. A. H. Guaitero, D. A. R. Perdomo, K. Kumaran, and G. R. Gao, “The supercodelet architecture,” in Proceedings of the 1st International Workshop on Extreme Heterogeneity Solutions, ser. ExHEET’22. New York, NY, USA: Association for Computing Machinery, 2022. [Online]. Available: https://doi.org/10.1145/3529336.3350823
[9] H. Edwards, C. R. Trott, and D. Sunderland, “Kokkos: Enabling manycore performance portability through polymorphic memory access patterns,” Journal of parallel and distributed computing, vol. 74, no. 12, pp. 3202–3216, 2014.
