Towards a Programming Paradigm for Reconfigurable Computing: Asynchronous Graph Programming

Joshua Fryer
Systems and Computer Engineering
Carleton University
Ottawa, Canada
joshfryer@cmail.carleton.ca

Paulo Garcia
Systems and Computer Engineering
Carleton University
Ottawa, Canada
paulo.garcia@carleton.ca

***Abstract*—The shift towards reconfigurable systems -hardware and software that adapt themselves to an external context-promises to unlock unprecedented performance, power consumption, and quality of service. However, reconfiguration imposes several challenges on the design of cyber-physical systems. Current design practices, including software frameworks and programming languages, are ill-prepared for supporting reconfiguration.**

In this paper, we explore Asynchronous Graph Programming, a programming paradigm and an associated model of computation designed for efficient and automated parallelization across processing elements, efficient reconfiguration (i.e., mapping of computational tasks across processing elements), and combining synchronous and asynchronous I/O handling within the same conceptual programming model. We also introduce an analytical model of parallelization, unlocked by Asynchronous Graph Programming, that can inform reconfiguration strategies.

We analyze the implications of our model through an analysis of reconfiguration scenarios given a program's characteristics; our analysis quantifies the benefits of reconfiguring software for higher levels of parallelism, given an amount of data left to process. We also introduce Horde, an open source Asynchronous Graph Programming interpreter, and use it to empirically validate the performance advantage of its automatic parallelism capabilities; in our experiments, automatic parallelization from one to four cores improves average case execution time by a factor of 2 and worst case execution time by a factor of 3.

***Index Terms*—Embedded software, Runtime environment, Parallel processing, Reconfiguration, Graph Processing, Programming languages**

I. Introduction

Modern cyber-physical systems encompass a broad spectrum of applications, from edge devices in cloud computing [1] to autonomous vehicles [2]. These applications are increasingly characterized by two unique characteristics: unprecedented demand for performance and battery life [3]; and a shift from static implementations, where software and hardware remain unchanged throughout system lifetime, to dynamic implementations, that adapt themselves to the external context [4]. Fundamental properties of real-time systems, such as predictability and observability, remain, of course, as important as ever [5], [6], and harder than ever to guarantee [7], especially with many systems becoming more and more distributed [8].

This shift towards reconfigurable, adaptable systems is an inescapable need to guarantee tomorrow's performance and power consumption [9]: examples include new heterogeneous multi-core adaptable software architectures [10] and application-specific accelerators on reconfigurable hardware (FPGAs) [11]. The vast majority of modern cyber-physical systems are adaptable in some way [12], due to, e.g., market demand for upgrades [13].

Reconfigurability and adaptability, however, impose several challenges on the design of cyber-physical systems [14]. Current design practices, including software frameworks, programming languages and hardware artifacts, are ill-prepared for supporting reconfiguration, especially if these include evolving hardware [15], deployment across heterogeneous platforms [16], or performance-driven design [17].

In this paper, we describe a novel programming paradigm and its associated model of computation, and how its unique properties allow it to implement reconfigurable, adaptable software. Specifically, this paper offers the following contributions:

- We describe the semantics of Asynchronous Graph Programming, a paradigm for parallel, heterogeneous, asynchronous computing.
- We describe its model of computation, as well as the runtime engine that implements such model, showing how it supports reconfiguration at multiple levels.
- We formulate a model that quantifies the performance speedups from reconfiguration for parallelism, i.e., for deploying a program across other available processing units, in function of how much of that program must still be evaluated.
- We demonstrate an empirical evaluation of such model, based on the current implementation of our Asynchronous Graph Programming paradigm.

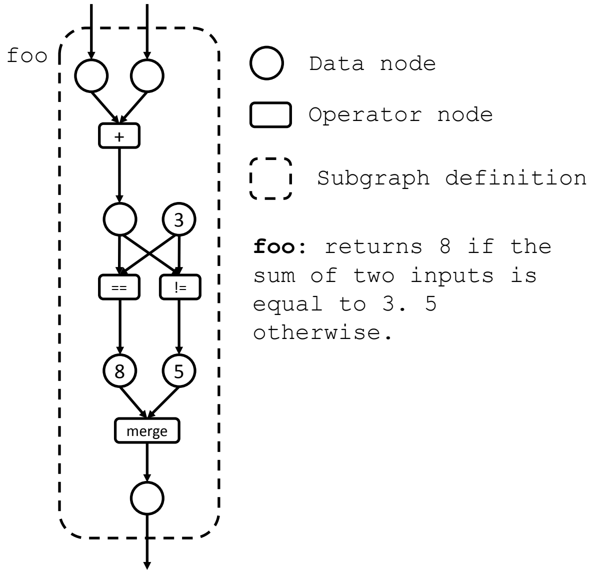

foo
+
3
==
!=
8
5
merge
Data node
Operator node
Subgraph definition
foo: returns 8 if the sum of two inputs is equal to 3. 5 otherwise.

Fig. 1. Example program in AGP semantics.

II. The Asynchronous Graph Programming paradigm

Asynchronous Graph Programming (AGP) is a programming paradigm and an associated model of computation designed for achieving three main objectives:

1) Efficient and automated parallelization across homogeneous and heterogeneous processing elements.
2) Efficient reconfiguration (i.e., mapping of computational tasks across processing elements).
3) Combining synchronous and asynchronous I/O handling within the same conceptual programming model.

AGP achieves the first two through its model of computation, and the third by its semantics, i.e., its conceptual programming model.

A. AGP semantics

AGP is technically a type of value-level programming. In AGP, a directed graph defines a *program*: every node in the graph represents either an operation on data, or a potential single assignment datum in that program. Every data node is said to be *constructed* if its value is known (either by defining it at compile time, or by calculating it at runtime), *unconstructed* if its value is not yet known, or *destroyed* (removed from program) if it is determined that it can never be constructed. Once a node is assigned a value, it does not change.

A particular *execution* of a program, for a given input stream, corresponds to a particular set of constructed and destroyed nodes (and respective values if constructed), such that the effectively constructed graph is a subset of the original AGP graph, which is a superset of all possible executions of a program. AGP semantics support this behavior by testing for construction. E.g., boolean tests (if-else) will construct either a datum *true* or a datum *false*, but not both. In a program definition, both these data will be defined as unconstructed data. Upon execution, one of these data will be constructed and the other destroyed. Subgraphs that depend on the constructed data will then be evaluated: subgraphs that depend on the destroyed data will be destroyed as well (pruned).

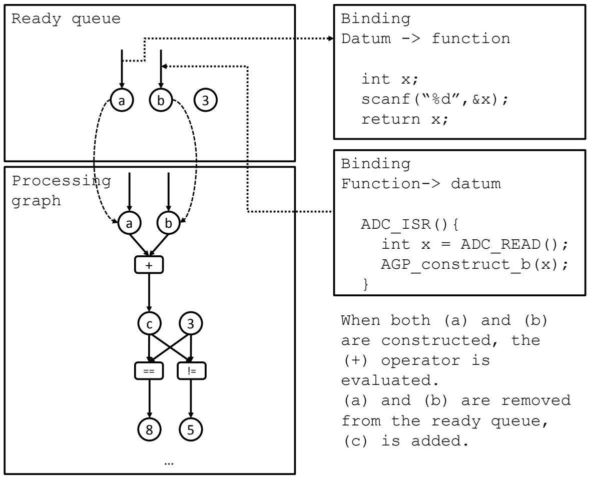
Whilst there are similarities between AGP and both functional and dataflow programming, there are key fundamental differences. Dataflow (e.g., CAL [18]) implements recursion through feedback networks, assuming mutable state. Unlike dataflow, AGP preserves immutability: recursion is achieved through dynamic graph expansion. Subgraphs (the AGP modularization and encapsulation approach) may create dynamic copies of themselves, if all data required for input is constructed (data destruction is the base case to terminate recursion). Unlike pure functional programming, where state and I/O must be implemented through monadic operations (e.g., Haskell [19]), AGP implements input and output as first class operators: both are represented by a graph edge with an empty end (directionality determines input or output). The example program depicted in Fig. 1 illustrates the main features of AGP semantics.

By leveraging I/O as a first class operation and execution that depends on construction, the AGP paradigm combines synchronous and asynchronous I/O processing. Programmers do not need to concern themselves with the order of input processing nor race conditions that may arise from shared state: simply, programmers specify the data dependencies (which dictate processing) in function of the existence, or non-existence, of inputs. For example, if two sources of inputs are present (e.g., synchronous user input and a hardware interrupt), an AGP program defines behaviors for interrupt occurrence and non-occurrence as mutually exclusive sub-graphs. Should the interrupt occur before the respective subgraph is evaluated, its corresponding datum will be constructed and fed into the computation. Should it not be created, its subgraph will be pruned upon evaluation (and potentially restored later through a recursive call). Thus, race conditions are prevented by construction, and the programmer's conceptual view needs not distinguish synchronous and asynchronous processing.

B. Model of computation

AGP's runtime engine implements its model of computation: i.e., the runtime engine is responsible for processing an AGP program, obeying its semantics, and mapping I/O operations to external data sources/sinks. The AGP runtime engine maintains an immutable graph that specifies the program (i.e., the *code* graph) and creates a mutable copy that it operates upon for processing (i.e., the *processing* graph - this is necessary as recursive programs must create copies of sub-graphs: the runtime engine must always keep an original copy of every sub-graph).

AGP graphs are processed by *N* ready queues, where *N* is the number of parallel processing elements (e.g., CPU cores: we give additional details on parallelization below). Each



```
Binding
Datum -> function

  int x;
  scanf("%d",&x);
  return x;
```

```
Binding
Function-> datum

  ADC_ISR(){
    int x = ADC_READ();
    AGP_construct_b(x);
  }
```

When both (a) and (b) are constructed, the (+) operator is evaluated.
(a) and (b) are removed from the ready queue, (c) is added.

Fig. 2. AGP ready queue (model of computation) and associated I/O mappings in both directions. In this example, one input is bound to a function called upon datum evaluation (user input) whilst another is bound to a hardware interrupt (ADC).

ready queue contains a list of all data that can be evaluated at the current state; upon program initialization, these are only constants and input operators. As data are constructed, they are added to the ready queue, and any data that have been completely processed (meaning, all data that depend on them have been constructed) are removed from the queue. Thus, a program is processed by following the flow of data dependencies in such program. The ready queue is guaranteed to analyze only data that can conceivably be constructed at the current state.

C. I/O mapping

The runtime engine is responsible for mapping AGP I/O operators to data sources/sinks. If the runtime engine is deployed bare-metal, this is achieved by low-level ”hooks” in the engine’s code that are linked to processing graph’s nodes when a program is loaded (bi-directionally). E.g., interrupt service routines (ISRs) are part of the runtime engine’s code base. Upon AGP program compilation, if an input source is mapped to a timer interrupt, then the runtime engine populates the timer ISR hook with a call to the corresponding input datum in the processing graph, essentially *binding* the ISR to the construction of a datum which is in turn added to the ready queue. Synchronous input (e.g., reading user-inputted data from a console) is bound in the reverse way: a datum is added to the ready queue as soon as it is possible to evaluate it, and when the runtime engine processes it, it calls a low-level function (e.g., *scanf()*) to construct it. Hence, the details of I/O mapping are abstracted from the graph processing aspects of the runtime engine.

D. Parallelization

AGP supports efficient parallelization across heterogeneous processing elements by transforming the problem of code parallelization into a problem of graph partitioning. Notice that every edge in an AGP graph corresponds to data processing and transfer between two memory positions. At the low level (optimizations not-withstanding), this corresponds to data read from memory (from a graph node) to the processor, processing as per the operator, and writing the data to a new memory position (destination graph node) for construction. Memory as the bottleneck of performance is a well-established fact [20]: optimizing performance means minimizing the number of memory accesses. Because cache memories are already quite efficient at doing so on a per-core basis, the performance bottleneck in a multi-core parallel system is shared memory [21]: we want to minimize data transfers between the processing functions on each core. In AGP, the problem of multi-core parallelization can be thus formulated as: given *N* processing elements, we want to partition a processing graph into *N* subgraphs, such that the number of elements in each subgraph is approximately the same, and we minimize the number of edges between subgraphs. Whilst we have identified some suitable algorithms for performing this automated parallelization (e.g., [22]) we have not yet empirically verified their suitability and limitations. The runtime engine deploys one ready queue per core, which operates only in memory destined for its subgraph, thus making efficient use of caches.

E. Reconfiguration

AGP supports dynamic reconfiguration and adaptability by enabling efficient execution of three reconfiguration tasks:

- Changing the number of parallel processing elements at runtime. Efficient allocation of a graph from *1* to *N* processing elements can be pre-computed offline. At runtime, based on operational constraints and context (e.g., load processing requirements) parallelization can be increased by simply copying partial ready queues, according to the allocation strategies, from one core to another, without significant pressure on instruction caches, as would happen in imperative languages where substantial amounts of code must cached when re-allocating threads.
- Re-mapping sub-graphs to processing elements (including heterogeneous) at runtime. Partial or complete programs can be moved from one core to another, following the same strategy described before, by simply copying ready queues (which impact data caches only). This is especially useful in situations where different cores have different power profiles (e.g., high-performance/high-power versus low-performance/low-power) and adaptation occurs in function of battery life constraints. Because only data must be transfered, this process is seamless even if different cores implement different instruction sets: the runtime engine must be compiled for each heterogeneous instruction set, but re-mapping of AGP programs occurs seamlessly.
- Hot-swapping of hardware I/O sources/sinks at runtime. Should we wish to re-map the source of data at runtime, e.g., in cases of hardware redundancy to account for runtime faults, where a backup system comes online when a primary system fails, AGP can re-map sources/sinks of

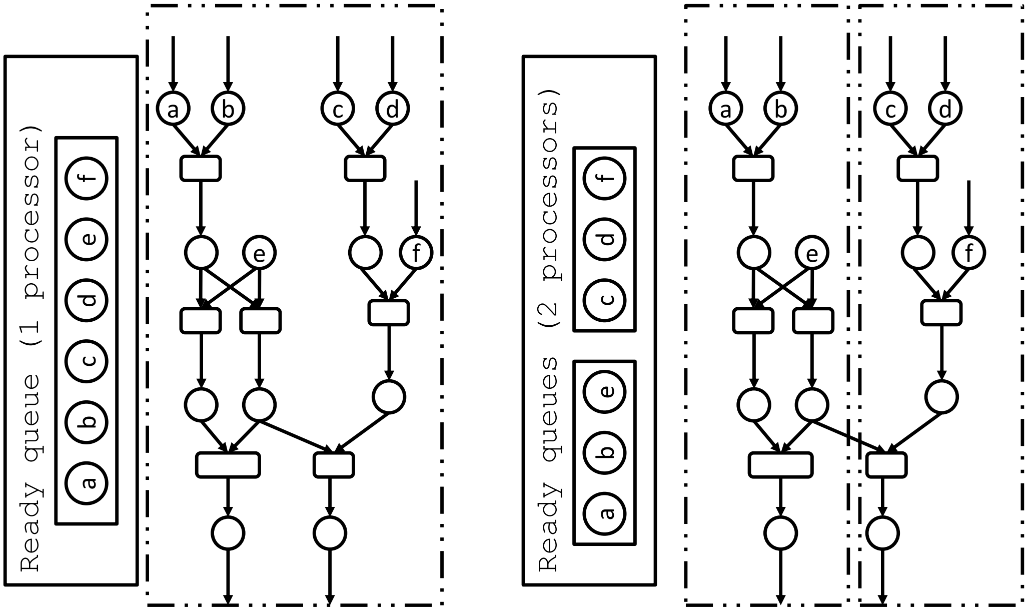


Fig. 3. AGP model of parallelization. In this example, parallel processing across two processors, based on a graph partitioning that divided the graph into two sub-graphs with approximately the same number of nodes, minimizing edges across sub-graphs (one edge).

data by modifying the low-level hooks, re-binding data to a new system. This separates reliability concerns (non-functional) from algorithmic concerns (functional): AGP programs do not need to account for hardware changes, delegating that responsibility to the runtime engine.

III. Reconfiguration Model

Depending on an application's primary concerns, different reconfiguration strategies may apply, prioritizing different aspects of a system's execution. In this paper, rather than proposing new or evaluating extant reconfiguration strategies, we focus on modeling AGP's reconfiguration capabilities so practitioners/researchers of reconfiguration strategies can use these models to inform their work.

We are interested in modeling the overhead (in execution time) associated with changing the number of processing units responsible for processing a graph, i.e., dynamically changing the number of parallel ready queues, as this informs performance-driven reconfiguration algorithms. We are also interested in modeling the amount of data processed per processing unit in the same scenario, as this has a substantial impact on data caches, again impacting performance. These variables (degree of parallelism, overhead for ready queue reallocation, and data size overhead) can be composed with total execution time and data size for a full program running on one processing unit to derive a function that estimates total execution time in function of degree of parallelism. Notice that we do not encompass any code overhead: it is assumed that the runtime engine is already deployed on all processing units, and there is no need for code transfers when runtime reconfiguration occurs.

We define T_n as the total execution time for a program across n processing units, such that the total execution time for that program on a single processing unit is T_1. T_n is defined as the processing time if a program is allocated to n processing units from startup, having pre-computed allocation of ready queues offline. We define the partial time Tp_d as the time required to compute a partial program (a subgraph) consisting of d nodes on a single processing unit, such that, if a graph of D data is uniformly partitioned in an idealized scenario:

$$T_n = Tp_{\frac{D}{n}} \tag{1}$$

We define O_q as the overhead (in time units) caused by transferring a (full or partial) ready queue from one processing unit to another, such that $O_q = f(q)$ and q is the number of data in the queue to be transferred; in other words, such that the overhead is a function of the number of data in the queue. We assume that O_q includes the overhead caused by populating a new core's data cache with the subgraph associated with the ready queue data. The function $f(q)$ must be adapted to the specific hardware details: e.g., for a system with processor cores accessing a shared memory without caches, $f(q)$ can likely be defined as linear function, such as $f(q) = t \times q$, where t is the time required to transfer a single datum across memory positions. For systems with cache memories, distributed memories or any form of Non-Uniform Memory Access (NUMA), $f(q)$ is likely not a linear function and must be determined accordingly.

If, at a point in time during a program execution, such that there are D' data left to evaluate, such that $D' < D$ (assuming that recursive computations that dynamically create more data are already accounted for in both D and D'), and the remaining program can be uniformly partitioned in an idealized scenario, the total time for program execution when

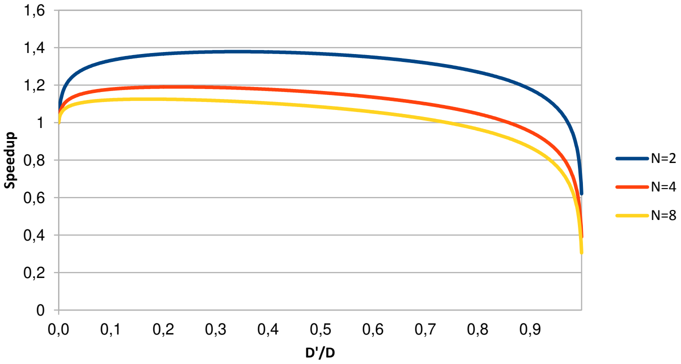


Fig. 4. Speedup (new over original time - lower is better) in function of reconfiguration at different program execution times (percentage of data left to process in parallel) for both O_q and Tp of complexity $\mathcal{O}(\log n)$.

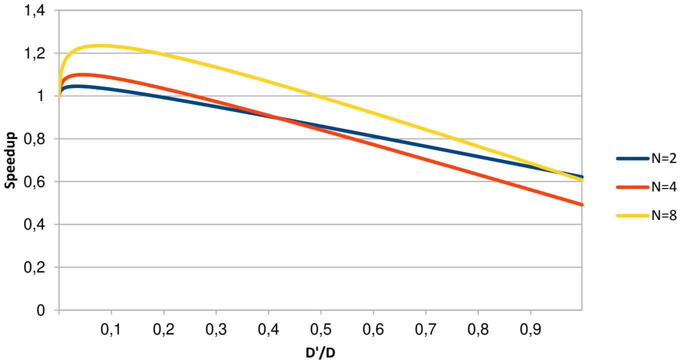


Fig. 5. Speedup (new over original time - lower is better) in function of reconfiguration at different program execution times (percentage of data left to process in parallel) for O_q of complexity $\mathcal{O}(\log n)$ and Tp of complexity $\mathcal{O}(n)$.

reconfiguring from one to N processing units, ${T_n}'$, can thus be defined as:

$$ {T_n}' = Tp_{D-D'} + (O_q \times N) + Tp_{\frac{D'}{N}} \qquad (2 $$

I.e., ${T_n}'$ is the time required to process $D - D'$ on th original, single processing unit ($Tp_{D-D'}$) plus the time re quired to transfer the partial ready queue across each new processing unit ($O_q \times N$) plus the time required to process $\frac{D}{N}$ data ($Tp_{\frac{D'}{N}}$). Equation 2 equals Equation 1 when $D = D$ i.e., $T_n = {T_n}'$ when all the parallelization is performed offline prior to program execution.

Estimating the growth rate for O_q and $Tp_{\frac{D'}{N}}$ (both of whic are implementation dependent) in function of D' and q, as well as the relationship between D' and q (which is program dependent) can inform reconfiguration strategies, e.g., ideal number of processing units for any given program state; such study is, however, outside the scope of this paper and reserved for future work, although we provide a superficial analysis in the next section. It should be noted that this framework provides a way of quantifying the degree of parallelism for a program, as a function of the growth rates of different model components.

IV. Experiments and Results

A. Model analysis

AGP's analytical reconfiguration model provides a way to analyze degrees of parallelism to inform reconfiguration. Whilst we have not yet performed a thorough analysis, we present a small evaluation of how an AGP program behaves given different growth rates for the functions O_q and Tp.

As D' increases (towards a higher and higher percentage of D), O_q and $Tp_{\frac{D'}{N}}$ increase whilst $Tp_{D-D'}$ decreases (obviously, $Tp_{\frac{D'}{N}}$ also decreases as N increases). Analyzing Equation 2, one can see that whether ${T_n}'$ is greater or smaller than T_1 depends on the growth rates for each term.

If O_q is the fastest growing term, there is an upper bound on the value of D', past which parallelism hurts performance: the overhead of transferring data outweighs the advantages of parallel performance (this upper bound is also dictated by N, but it is guaranteed to exist).

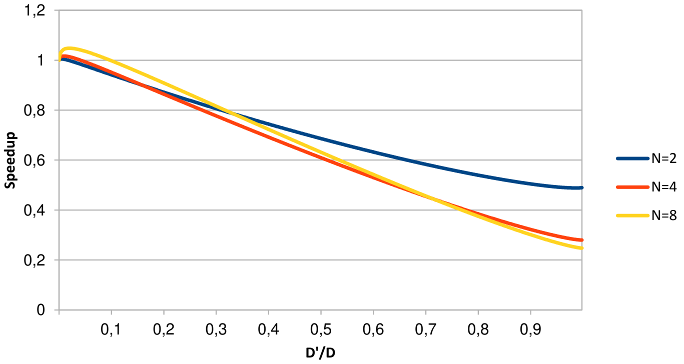


Fig. 6. Speedup (new over original time - lower is better) in function of reconfiguration at different program execution times (percentage of data left to process in parallel) for O_q of complexity $\mathcal{O}(\log n)$ and Tp of complexity $\mathcal{O}(n \log n)$

If O_q is not the fastest growing term, there is a lower bound on the value of D' past which reconfiguration for parallelism brings performance advantages; these are the particularly interesting scenarios. We simulated synthetic data for an implementation of O_q of complexity $\mathcal{O}(\log n)$, compared to three implementations of Tp of complexities $\mathcal{O}(\log n)$, $\mathcal{O}(n)$ and $\mathcal{O}(n \log n)$: these are plotted, for different values of N, in Figures 4, 5 and 6, respectively, as *speedup* ($\frac{{T_n}'}{T_1}$) in function of percentage of data processed in parallel ($\frac{D'}{D}$).

These results are illustrative of classes of functions that represent O_q and Tp. When both have complexity $\mathcal{O}(\log n)$ (Fig. 4), i.e., they grow at the same rate, performance worsens (execution time ratio grater than 1) for low values of D': overhead of transferring data overcomes performance advantages of parallelism. As D' becomes a bigger and bigger portion of total data, we start to see performance improvement, and this

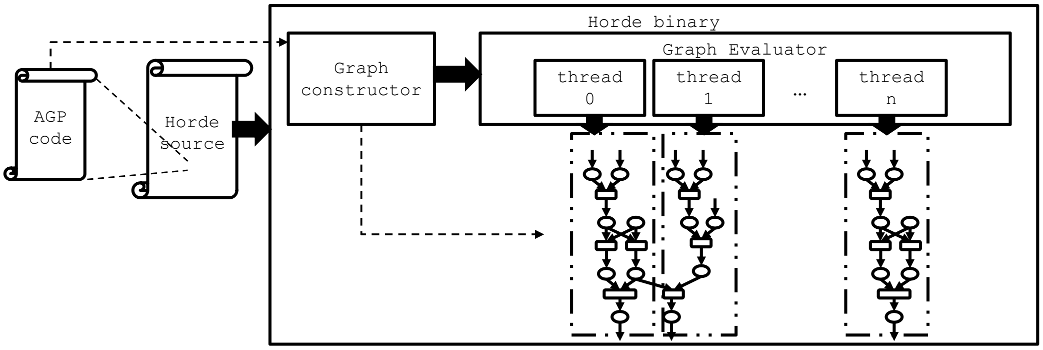


Fig. 7. The Horde interpreter architecture. AGP code is implemented as an embedded language that is expanded into graph construction.

happens more quickly for higher values of N. The specific values for O_q and Tp dictate when the function crosses speedup = 1, and the value of O_q dictates the value of the function at $D' = D$; i.e., how much higher the real value is than the theoretical limit of $\frac{1}{N}$. E.g., in Fig. 4 the function for N=2 intersects $D' = D$ at ≈ 0.6 (rather than the theoretical limit at 0.5).

As Tp grows more quickly than O_q, we start seeing that performance improves for lower values of D'. In Fig. 5, where the growth of Tp is $\mathcal{O}(n)$, we see that for small values of D' parallelization across fewer parallel computational units results in increased performance more quickly: this makes intuitive sense, as we minimize data transfer overhead where little parallelism can be achieved. As the value of D' increases, higher values of N eventually result in higher performance gains (e.g., N=4 surpasses N=2 at $\frac{D'}{D} \approx 0.4$ and N=8 surpasses both N=2 and N=4 at $\frac{D'}{D} \approx 0.9$).

As Tp grows even more quickly, such as in Fig. 6, where the growth of Tp is $\mathcal{O}(n \log n)$, a similar behaviour occurs even more rapidly: N=2 initially outperforms higher levels of parallelism, but is more quickly surpassed by N=4 and N=8. Whilst we have much deeper analyses and empirical validations to perform, this model provides a language for reasoning about parallelism and reconfiguration in quantifiable terms.

B. Empirical results

We conducted all experiments on a machine equipped with a quad-core i5-7440HQ CPU @2.80GHz, with 8GB of RAM, running Ubuntu 16.04LTS. Our experiments were conducted on the current prototype of the AGP compiler and runtime engine: the Horde interpreter (publicly available in open-source form here[1]). At present, Horde does not yet support bare-metal deployment nor associated interrupt to datum binding; all I/O is mapped to user (file) input and output. However, Horde already supports the full AGP semantics and model of computation, as well as parallelization (implemented through multi-threaded execution).

The current version of the Horde interpreter was primarily built as a testing environment for the AGP semantics; hence, it is not optimized for performance. AGP graphs are implemented as multiply-linked linked lists which the interpreter reduces throughout execution. The ready queue is also implemented as a linked list that points to nodes in the program. AGP code is programmed as an embedded language, i.e., based on macro expansions that generate interpretable code. Fig. 7 provides a high level overview of the architecture of Horde. AGP code is not programmer-friendly, nor meant to be: it is meant as an intermediate representation that higher level languages can build upon. Despite its immaturity, Horde already provides a complete environment to test AGP semantics, and can be used to experiment with software runtime reconfiguration. Horde supports parallel execution of multiple programs (one program per thread), or parallel execution of one program across several threads in a master-slave fashion: one master thread initially has exclusive access to all nodes in the program, and can transfer ready nodes to slave threads.

We began by evaluating the overhead required to transfer ready nodes from a master to a slave thread. We created a program with approximately 100 data (200 nodes total, data and operators), where all 100 operators are ready at startup. We then tested transferring $q = \{1..100\}$ as soon as the master thread starts executing and measured the processor time required to transfer the data. For every value of q, we ran Horde 100 times, for a total of 10,000 executions. We calculate processor time through pthread API functions. Results of this experiment are displayed in Fig. 8. Interestingly, despite the function that transfers ready nodes having complexity N (it must traverse the entire graph, copying the nodes that have been marked by the master thread), we do not observe a constant execution time: there are peaks between $q \approx 9$ and $q \approx 45$. This confirms the hypothesis in Section III that O_q is implementation dependent: in this case, we hypothesize that for these values of q, cache behavior, particularly due to inter-thread interference, causes significant delays, highlighting the need to perform architectural (implementation) profiling for efficient reconfiguration.

We then assessed the performance gained through runtime

[1] https://github.com/paulofrgarcia-carleton/Horde-Public-release-

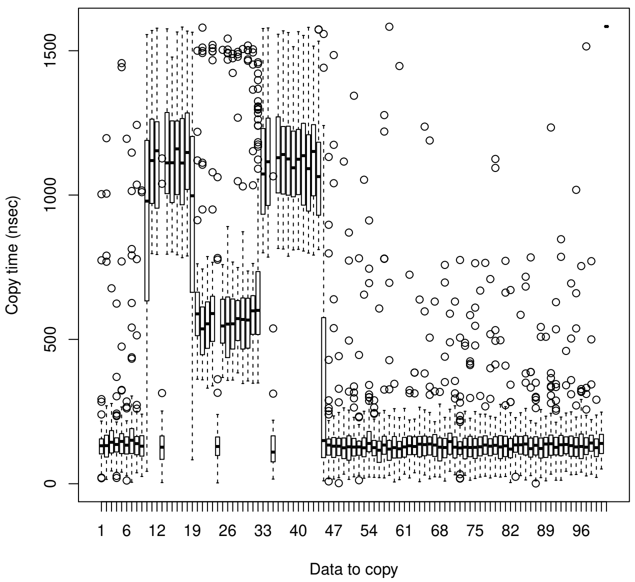


Fig. 8. Data transfer time (nsec) across threads for different number of ready nodes in the Horde interpreter.

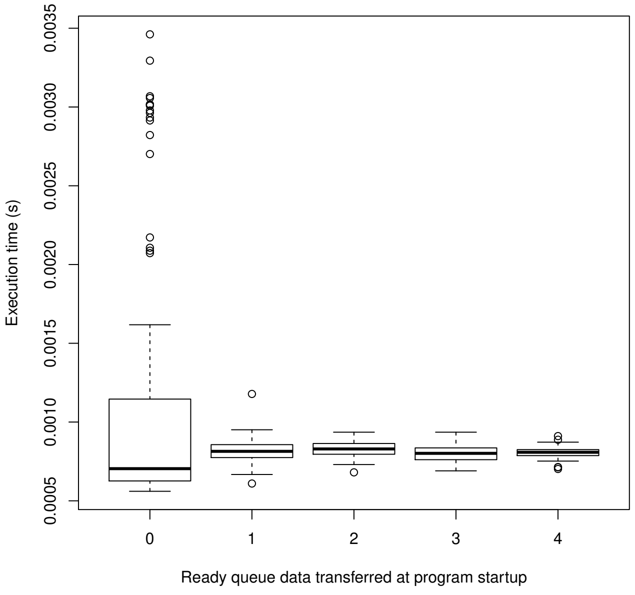


Fig. 9. Total execution time (sec) for dual-thread execution for different number of transferred ready nodes in the Horde interpreter.

reconfiguration for parallelism. We created a dual thread program, processing a math-intensive application (calculating the factorials of eight input data). Upon startup, we experimented with transferring $q = \{0..4\}$, i.e., from single-thread execution up to fully balanced parallel execution. For each value of q, we ran Horde 100 times, for a total of 500 executions: results are displayed in Fig. 9. It's easily observed from Fig. 9 that the average execution time follows a decaying exponential; best-case execution time behaves as our models suggest, albeit there is not enough data to confirm the specific growth rate. Interestingly, execution time variation decreases as the load is balanced more and more equally across the different threads, suggesting that, in this case, parallel execution likely takes better advantage of the memory hierarchy (probably due to latency hiding by the scheduler).

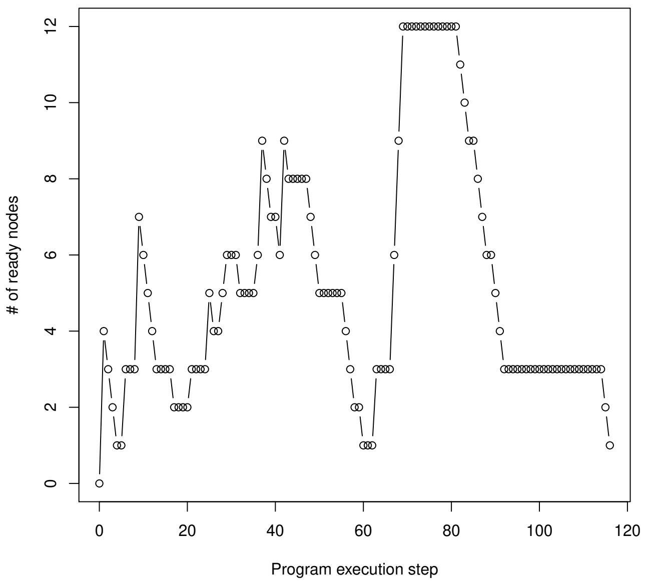


Fig. 10. Number of ready nodes for the execution of *SolveCubic* in the Horde interpreter. Peaks correspond to calculating all required inputs to an equation, which then collapse as the result is calculated.

Finally, we implemented one example from the MiBench suite [23] (*basicmath* from the *automotive* package) in Horde to analyze how the ready queue behaves in a real representative program (we will eventually implement the entire suite for comparison). Fig. 10 shows the evaluation of the ready queue as the program is evaluated (one call to *SolveCubic*, 117 steps in this implementation for this input data).

V. Conclusions and Future Work

Modern cyber-physical systems are now characterized by unprecedented demand for performance and battery life, and a shift from static implementations, where software and hardware remain unchanged throughout system lifetime, to dynamic implementations, that adapt themselves to the external context. Reconfigurability and adaptability, however, impose several challenges on the design of cyber-physical systems. In this paper, we described Asynchronous Graph Programming, a novel programming paradigm (and its associated model of computation), designed for reconfigurable embedded software.

AGP semantics provide a way to express the degree of parallelism of a program: we have introduced an analytical model that captures both program-specific features and architecture/implementation-specific features that can inform reconfiguration strategies. The AGP runtime engine implements AGP semantics, whilst providing mechanisms for efficient handling of common real-time embedded system requirements (e.g., asynchronous event handling). We have empirically validated AGP semantics, and partially validated our analytical model, through experimentation with Horde, an open-source AGP interpreter.

Future work will focus on three fronts. First, the development of code generators for bare-metal deployment of AGP programs, which will allow us to conduct more in-depth empirical evaluations. Second, the refinement and further validation of our analytical model, informed by the empirical results, and its applicability in context-aware reconfiguration strategies. Third, the semantics of high level languages that embedded the AGP paradigm, towards the maturity of design frameworks, supported by hardware and software artifacts, that will unleash the potential of reconfigurable computing.

Acknowledgment

We would like to thank Carleton University's I-CUREUS program for undergraduate student research for sponsoring Joshua Fryer's research work.

References

[1] S. Meixner, D. Schall, F. Li, V. Karagiannis, S. Schulte, and K. Plakidas, "Automatic application placement and adaptation in cloud-edge environments," in *2019 24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE, 2019, pp. 1001–1008.

[2] S. Mubeen, "Developing predictable embedded systems in the vehicle industry: Results and lessons learned," in *IEEE International Conference on Industrial Technology (IEEE ICIT), FEB 13-15, 2019, Melbourne, AUSTRALIA*. IEEE, 2019, pp. 1063–1065.

[3] H. Zhang and H. Hoffmann, "Maximizing performance under a power cap: A comparison of hardware, software, and hybrid techniques," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 2, pp. 545–559, 2016.

[4] S. Gaur, L. Almeida, E. Tovar, and R. Reddy, "Cap: Context-aware programming for cyber physical systems," in *2019 24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE, 2019, pp. 1009–1016.

[5] M. García-Gordillo, J. J. Valls, and S. Sáez, "Heterogeneous runtime monitoring for real-time systems with art2kitekt," in *2019 24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE, 2019, pp. 266–273.

[6] A. Hasanbegović, M. Ventovaara, J. Wiklander, and S. Mubeen, "Optimising vehicular system architectures with real-time requirements: An industrial case study," in *IECON 2019-45th Annual Conference of the IEEE Industrial Electronics Society*, vol. 1. IEEE, 2019, pp. 4501–4508.

[7] V. Lesi, Z. Jakovljevic, and M. Pajic, "Synchronization of distributed controllers in cyber-physical systems," in *2019 24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE, 2019, pp. 710–717.

[8] A. Cenedese, M. Frodella, F. Tramarin, and S. Vitturi, "Comparative assessment of different opc ua open–source stacks for embedded systems," in *2019 24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE, 2019, pp. 1127–1134.

[9] T. Guillaumet, A. Sharma, E. Feron, M. Krishna, R. Narayan, P. Baufreton, F. Neumann, and E. Grolleau, "Using reconfigurable multi-core architectures for safety-critical embedded systems," in *2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC)*. IEEE, 2016, pp. 1–6.

[10] B. Donyanavard, T. Mück, S. Sarma, and N. Dutt, "Sparta: Runtime task allocation for energy efficient heterogeneous manycores," in *2016 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS)*. IEEE, 2016, pp. 1–10.

[11] M. A. Aguilar, R. Leupers, G. Ascheid, and L. G. Murillo, "Automatic parallelization and accelerator offloading for embedded applications on heterogeneous mpsocs," in *Proceedings of the 53rd Annual Design Automation Conference*. ACM, 2016, p. 49.

[12] J. Otto, B. Vogel-Heuser, and O. Niggemann, "Automatic parameter estimation for reusable software components of modular and reconfigurable cyber-physical production systems in the domain of discrete manufacturing," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 1, pp. 275–282, 2017.

[13] F. Boschi, C. Zanetti, G. Tavola, and M. Taisch, "Functional requirements for reconfigurable and flexible cyber-physical system," in *IECON 2016-42nd Annual Conference of the IEEE Industrial Electronics Society*. IEEE, 2016, pp. 5717–5722.

[14] L. Liu, J. Zhu, Z. Li, Y. Lu, Y. Deng, J. Han, S. Yin, and S. Wei, "A survey of coarse-grained reconfigurable architecture and design: Taxonomy, challenges, and applications," *ACM Computing Surveys (CSUR)*, vol. 52, no. 6, pp. 1–39, 2019.

[15] M. Ghane, S. Chandrasekaran, R. Searles, M. S. Cheung, and O. Hernandez, "Path forward for softwarization to tackle evolving hardware," in *Disruptive Technologies in Information Sciences*, vol. 10652. International Society for Optics and Photonics, 2018, p. 106520O.

[16] J. Castrillon, M. Lieber, S. Klueppelholz, M. Völp, N. Asmussen, U. Assmann, F. Baader, C. Baier, G. Fettweis, J. Froehlich *et al.*, "A hardware/software stack for heterogeneous systems," *IEEE Transactions on Multi-Scale Computing Systems*, vol. 4, no. 3, pp. 243–259, 2017.

[17] M. Kreutzer, J. Thies, M. Röhrig-Zöllner, A. Pieper, F. Shahzad, M. Galgon, A. Basermann, H. Fehske, G. Hager, and G. Wellein, "Ghost: building blocks for high performance sparse linear algebra on heterogeneous systems," *International Journal of Parallel Programming*, vol. 45, no. 5, pp. 1046–1072, 2017.

[18] P. Garcia, D. Bhowmik, A. Wallace, R. Stewart, and G. Michaelson, "Area-energy aware dataflow optimisation of visual tracking systems," in *International Symposium on Applied Reconfigurable Computing*. Springer, 2018, pp. 523–536.

[19] D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières, "Flexible dynamic information flow control in haskell," in *Proceedings of the 4th ACM symposium on Haskell*, 2011, pp. 95–106.

[20] X.-H. Sun and Y.-H. Liu, "Utilizing concurrency: A new theory for memory wall," in *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 2016, pp. 18–23.

[21] V. Stegailov, G. Smirnov, and V. Vecher, "Vasp hits the memory wall: processors efficiency comparison," *Concurrency and Computation: Practice and Experience*, vol. 31, no. 19, p. e5136, 2019.

[22] K. Andreev and H. Racke, "Balanced graph partitioning," *Theory of Computing Systems*, vol. 39, no. 6, pp. 929–939, 2006.

[23] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Proceedings of the fourth annual IEEE international workshop on workload characterization. WWC-4 (Cat. No. 01EX538)*. IEEE, 2001, pp. 3–14.