DARTS: A RUNTIME BASED ON THE CODELET EXECUTION MODEL

by

Joshua Suetterlein

A thesis submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Master of Science in Electrical and Computer Engineering

Spring 2014

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 - 1346

DARTS: A RUNTIME BASED ON THE CODELET EXECUTION MODEL

by

Joshua Suetterlein

Approved: _____
Guang R. Gao, Ph.D.
Professor in charge of thesis on behalf of the Advisory Committee

Approved: _____
Kenneth E. Barner, Ph.D.
Chair of the Department of Electrical and Computer Engineering

Approved: _____
Babatunde A. Ogunnaike, Ph.D.
Dean of the College of Engineering

Approved: _____
James G. Richards, Ph.D.
Vice Provost for Graduate and Professional Education

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Over the past decade computer architectures have drastically evolved to circumnavigate prevailing physical limitations in chip technology. Energy consumption and heat expenditure have become the predominant concerns for architects and chip manufacturers. Previously anticipated trends such as frequency scaling, deep execution pipelines, and fully consistent caches in future many-core systems have been deemed unsustainable.

Current architectures are exhibiting new trends including simpler pipelines, lower frequencies, and scratch pad memories. Moreover, these architectures have an ever increasing number of cores. Many predict future architectures to contain thousands of heterogeneous cores on a single die.

With these radical shifts in architectures, current execution models are struggling to adequately scale in performance and newer metrics like energy consumption. The shortcomings of current models have caused some to look back to fine-grained execution models designed for parallelism like dataflow and EARTH. Using these models as inspiration, the Codelet execution model is an event-driven, fine-grained model designed to exploit parallelism while providing efficient mechanism for locality.

In the following, we present the Delaware Asynchronous RunTime System (DARTS), an implementation of the Codelet model. DARTS is a faithful implementation of the Codelet model, providing a vehicle to reason and further develop codelet ideas. It provides two levels of parallelism, event-driven codelets permitting fine-grained parallelism and invoked threaded procedures which ensures locality. Furthermore, the DARTS runtime is built on a reconfigurable abstract machine allowing DARTS to provide performance portability across both architectures and applications. In addition,

we provide an in depth analysis of DARTS and its underlying model running on off-the-shelf hardware. Utilizing two x86 machines (both Intel and AMD), we explore the overheads of the codelet model and its implementation using micro benchmarks. Furthermore, we demonstrate DARTS' performance for two benchmarks, matrix multiply and breadth first search. Leveraging these results, we aim to establish the Codelet model as a promising execution model for future many-core architectures via an efficient and well-designed runtime. The following summarizes the contributions of this thesis:

1. A specification for the Codelet execution model
2. DARTS: An accurate implementation of the Codelet model
   (a) Two levels of parallelism; fine-grained tasks and procedures intertwined by fine-grain synchronization
   (b) Adaptable abstract machine capable scaling for different applications and hardware
3. A two phase evaluation of DARTS and the Codelet execution model running on available off-the-shelf hardware accomplish by
   (a) Micro benchmarking the base primitives employed by DARTS to realize the Codelet model
   (b) Evaluating case studies on selected benchmarks representative of workloads of interest including matrix multiply and breadth first search

Chapter 1

INTRODUCTION

In order to further science, the high performance community is on an unending journey to achieving extraordinary levels of computing. Continuing in this effort, our next task is to achieve exa-scale performance. We face several daunting challenges requiring novel solutions in order to accomplish this goal. These challenges go beyond compute performance to achieving energy efficiency, resiliency, security, and more. The aim of this thesis is to provide an execution model and runtime to act as a platform for future research addressing these problems.

Historically, the use of an execution model, or program execution model (PXM), has become unfashionable leading to a schism between hardware and software architects. More recently however, this has changed due to the advent of the many-core era. In order to provide useful (and usable) systems, both hardware and software developers have realized the need for a unified view of the entire system. An execution model provides the abstraction and principles upon which the underlying system architecture and system software should be conceived, designed, and developed. The execution model provides the governing principles for the system design, operation, management, and application. This model is implemented not by any single component of a system; rather it is the sum of its parts. The execution model's reach spans the whole system, crosscutting all layers including the system architecture and software stack.

One crucial piece of an execution model's implementation is a runtime system. A runtime provides flexibility in satisfying a particular execution model's requirements by bridging the gap between software and hardware. This permits researchers to study different models on a single platform. The approach is not without caveats however. A runtime is only capable of satisfying certain hardware limitations. If the hardware

does not provide sufficient mechanism to a runtime, certain features of an execution model may not be feasible. Furthermore, there are many trade-offs in deciding which features of an execution model should be in software (implemented in the runtime) or hardware.

Future exa-scale machines will have thousands of cores per node [5, 42]. Moreover, the physics of these systems (fabrication and operation) have brought new metrics like energy efficiency and resiliency to the forefront. It is unclear whether current models will be capable of scaling to the parallelism available in future large scale systems, and currently lack sufficient facilities to address these new metrics. As such, we believe now is the time for a new execution model.

An execution model suitable for exa-scale must be highly scalable, energy efficiency, and resiliency. The objective of this thesis is to propose an event-driven execution model. Event-driven execution goes beyond control-flow or dataflow to executing task based on *events*. We believe an event-driven execution model can sufficiently provide the mechanisms to tackle these challenges. Moreover we provide a means of exploring the model on current architectures using a runtime and several benchmarks. The following summarizes the contributions of this thesis:

1. A specification for the Codelet execution model, an event-driven, fine-grained, multi-threading model aimed at scaling to future large scale heterogeneous systems.
2. Delaware Adaptive RunTime System (DARTS), a codelet runtime for shared memory x86 systems.
3. A two phase evaluation of DARTS and the Codelet execution model via micro benchmarks and two case studies.

We begin by illustrating the shortcoming of current models and describing past execution models which paved the way for the Codelet model in chapter 2. Next we introduce the Codelet execution model in chapter 3. Chapter 4 presents DARTS and several micro benchmarks. We continue our codelet exploration in chapter 5 with two case studies. Lastly we present the related work in chapter 6 and conclude in chapter 7.

Chapter 2

BACKGROUND

Over the past several years, we have successfully entered the peta-scale era using execution models which have evolved from sequential computers. The principal execution model has been based on the the von Neumann model which is comprised of a serial process running in a linear address space controlled by a single program counter.

Parallelism has been introduced by adding multiple execution units (i.e. cores). One (possibly outdated) means of categorizing parallel machines has been to use Flynn's taxonomy. The parallel architectures included in the taxonomy are Single Instruction Multiple Data (SIMD), Multiple Instructions Single Data (MISD), and Multiple Instruction Multiple Data (MIMD). The most significant of these are SIMD and MIMD, and can be furthered decomposed beyond Flynn's taxonomy [22]. Parallel systems characterized by SIMD or MIMD can be a distributed memory system (one which is comprised of multiple address spaces requiring some form of message passing) or a shared memory system (one with only a single address space requiring coordinated synchronization operations).

The effects of many architectural trends are present in current parallel systems. For example many systems exhibit SIMD parallelism using SSE directives to take advantage of vector processing. Moreover, the SIMD idea has evolved to SIMT (Single Instruction Multiple Threads) popularized by NVIDIA's GPU accelerators [33]. Shared memory MIMD systems leverage parallelism through some form of threading. The most popular are the OpenMP and POSIX threading models. Distributed memory MIMD systems primarily use MPI to coordinate the execution of processors operating in distinct memory addresses. It is common to find some combination if not all of these

parallel features in the today's fastest large scale systems (e.g. a system using MPI for inter-node communication and OpenMP for intra-node communication). Today's parallel systems are already heterogeneous exploiting various forms of parallelism. Future architectures are expected to continue this trend while exposing even greater amounts of parallelism to the programmer.

Knowing the scale of future architectures, many are searching for more effective means of combining both inter-node and intra-node parallelism. This task has proven quite difficult, and often requires significant rewriting of application code and a high level of programmer expertise [12]. Despite this trend many are clinging to current execution models, in hopes to maintain their application investments (time, code, money, and tool-chain). As such, these parties seek an evolutionary path to exa-scale. This approach has been deemed the MPI + X where X is preferably OpenMP [1]. This approach faces several challenges.

OpenMP was first released in 1997, and is supported by most modern compilers [40]. OpenMP is a standard which extends the C languages with pragma based directives supporting thread parallelism. Prior to 2004, OpenMP has been centered on leveraging loop parallelism in a fork/join execution model. More recently, OpenMP has extended the standard to include asynchronous tasks.

The OpenMP model has previously focused on exploiting loop parallelism, dividing iteration into chunks. Different policies are used to determine the way in which chunks are distributed to threads. One problem encountered with this approach is how parallelism is expressed and thusly executed. Loop based parallelism leveraging a fork/join model can easily lead to system load-imbalance (where resources, primarily compute elements, are underutilized). This happens when chunks (or iterations) are not uniform (requiring varying amounts of time to execute). In these instances, threads are stalled at join points waiting for the remaining threads to finish, leading to underutilization.

The uniformity of work per iteration is primarily application dependent. However, the latency of memory accesses and other contended hardware resources can

cause non-uniformity in even regular applications [26]. Non-Uniform Memory Access (NUMA) effects have been caused by the movement of the memory controller onto the same die as the CPU [34]. Multiple cores with multiple memory controllers make up a single address space via a point to point network, and a cache coherent protocol enables access to remote memory. Accessing memory attached to a processor clearly has a lower latency then accessing data through the network. The differing latencies of memory accesses can lead to very different execution times for OpenMP chunks. Moreover, we find that compute elements are fighting for shared resources like network bandwidth and floating point units. Both of these trends are leading to less uniformity in applications.

To mitigate these issues many are exploring asynchronous tasks including OpenMP. Unfortunately, many, including OpenMP, limit their exploration of tasks based on control-flow (as opposed to dataflow based execution). The dataflow model conveniently decomposes a problem to show the maximum parallelism in an application [2] which will be important for mapping applications to future systems.

Another important issue facing the MPI + X approach is the poor (or non-existent) interaction between the two distinct models. An unsatisfactory approach to combining these two models is to use two separate runtimes. This may lead to message buffers (within the runtimes) to be physically far (in the memory hierarchy) from producing and consuming threads leading to an increase in energy consumption as data must be moved [1]. A more satisfactory approach is explored in MPC [39], where both MPI and OpenMP standards are implemented in a single runtime. Unfortunately, despite a unified runtime, leveraging MPIs standard tends to forces workloads to be statically partition across nodes [32]. A unified model will be better poised to load-balance an entire system, ensuring efficient resource utilization. Moreover, an execution model for exa-scale must balance parallelism with locality to stay within an acceptable power budget. Addressing these issues in two separate but joined models is a daunting challenge.

To overcome these issues a unified, event-driven, fine-grained execution model

is needed. The required model's implementation should have low overhead (in order to scale), and permit fine-grain parallelism all while ensuring locality. Moreover, the model must embrace the heterogeneous resources available in future systems. The remainder of this section describes the path leading up to the Codelet execution model, a disruptive technology intended for exa-scale performance. We begin in section 2.1 by describing dataflow in several flavors and the EARTH execution model in 2.2.

2.1 Dataflow

The Codelet model's deepest roots lie in the dataflow model of computation. Dataflow provides several important features including fine-grain synchronization, functional programming, composability, and determinate execution [46]. Most importantly, dataflow exposes the maximum amount of parallelism available in a given program [2]. In order for an execution model to scale to an architecture with a larger number of cores, it must expose and exploit sufficient parallelism in underlying programs to fully utilize the system. For this reason, the dataflow model serves as a solid foundation for the exploration of an exa-scale capable execution model.

2.1.1 Static Dataflow

In the early 1970's, Jack Dennis proposed the first version of the dataflow model, later dubbed static dataflow [18, 19]. The dataflow model takes a radically different approach to organizing and executing instructions. Rather than expressing a program in a series of sequential instructions, programs are represented by graphs.

2.1.1.1 Dataflow Graphs

In dataflow, computation is modeled as a DataFlow Graph (DFG). A DFG is a directed graph consisting of nodes, arcs, and tokens [13, 31]. The following is a description of each component of a DFG:

Actor is a node which represents an operation to be performed on a piece or pieces of data. An actor will take one or more inputs, perform the specified operation, and provide data to its outputs.

Arc represents a data dependency between actors. Arcs signify the passing of data from one actor to the next, primarily representing a producer/consumer relationship.

Token represents data traveling on an arc between actors.

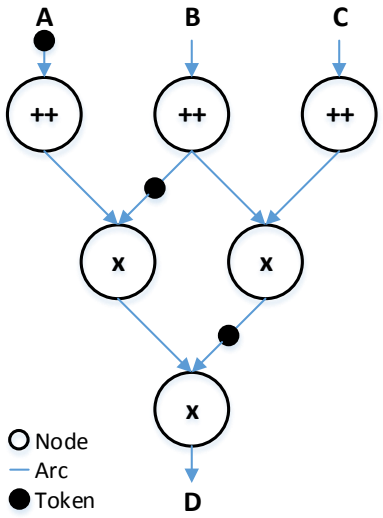


Figure 2.1: A DataFlow Graph (DFG) contains actors, arcs, and tokens.

2.1.1.2 Operational Semantics

Tokens passed from one actor to the next are used to signify the presence of data. The directed arcs represent the data dependencies between actors. An actor fires, or executes its operation, based on the following rule [19]:

General Firing Rule: An actor becomes enabled when a token is present on each of its input arcs. An actor then fires by consuming the input tokens, performing its operation, and placing a token, the result of the operation, on each of its output arcs.

As each actor fires, the execution of the dataflow program is progressed. Each individual actor has no internal state since the operation it represents is of the finest granularity. The state of the application instead is found in the entire DFG. A snapshot

of a DFG shows tokens traveling from actor to actor across arcs during execution. Thus, the state of a dataflow program is the sum of its actors, arcs, and tokens for a given snapshot.

Multiple actors are permitted to fire as long as the firing rule for each individual actor is met (since DFGs have no concept of the hardware executing them). This is the way in which parallelism is expressed; asynchronous actors are only bound by their inputs and execute concurrently. Moreover, since each actor is only a single operation limited only by its inputs, we see a DFG permits the finest granularity of parallelism.

2.1.1.3 Limits of Static Dataflow

Static dataflow differs from other incarnations of the dataflow model based on one simple restriction of the general firing rule. In static dataflow, an actor may only fire when all of its input tokens are available and all of its output arcs are empty. This important distinction lends the bases for static dataflow's other name, single-token-per-arc dataflow as ensuring output arcs' freedom before firing limits all arcs to containing at most one token. To ensure this property, network traffic is effectively doubled since actors must signal their preceding actors that their output arcs are free [4, 46]. This inefficiency led to the development of the dynamic dataflow model.

2.1.2 Dynamic Dataflow

Dynamic dataflow, also known as tagged-token dataflow builds on static dataflow addressing the overhead caused by the only permitting a single-token-per-arc [3, 48]. In dynamic dataflow, each token is augmented with a color which is used to determine for which instruction the data value is destined and additional contextual information used to determine which other tokens should be used when the instruction fires. By "tagging" tokens with a color, multiple tokens may reside on a single arc with a modification to the firing rule.

Tagged-token Firing Rule: An actor becomes enabled when tokens with corresponding colors arrive on all of its input arcs. The actor fires by removing the colored

tokens from its input arcs, executes the instruction, and places the result on its output arc with an appropriate color.

With multiple tokens per arc, data can be reordered as long as the firing rule is observed providing better performance. The trade-off of tagging tokens is the time required to match tokens. Depending on the operation and the number of tokens on an arc, locating matching tokens might be more costly than performing the actor's operation. For this reason, a fully associative memory is ideal for reducing the latency of token matching. Unfortunately a required associative memory is impractical due to its size. One alternative to overcoming this challenge is to use hashing techniques [46]. A more satisfactory method is to employ the explicit token store technique [36]. Token matching becomes extremely expensive when executing loops in dynamic dataflow since iterations may execute out of order. The explicit token store alleviates this burden by assigning each active iteration a frame. Actors leverage explicit token store by using a frame and offset to store and access each token.

2.1.3 Argument Fetching

One limitation of the previously mentioned forms of dataflow is excess storage and the network traffic required to copy operands. Argument-fetching dataflow attempts to address this shortcoming by dissociating the signal and data sent to consuming actors [23]. Alternatively, each actor contains an instruction, addresses to its arguments, an address to the result, addresses of actors to signal, an enable count, and a reset count. When an actor is fired, its arguments are read from memory and the result is written to memory. When the instruction is finished, the proceeding actors are signaled decrementing their enable count. Once the enable count reaches zero, an actor is fired, after which the counter is reset. By disassociating the signal and the data, the result only needs to be written and stored once. This trades time for space, as the latency of accessing arguments is increased.

2.2 EARTH

While dataflow exposes the maximum parallelism found in an application, it is still plagued with several drawbacks. One primary weakness of dataflow is the high cumulative overheads resulting from fine-grained synchronization. These costs may include reading and writing operands, signaling, and locating and executing enabled actors. Moreover these overheads are exacerbated when running a sequential code, where a von Neumann counterpart excels. Dataflow also fails to effectively take advantage of locality, which is a primary concern for future architectures as data movement dominates the energy consumption of a machine. Due to these limitations, hybrid von Neumann/dataflow architectures have been developed.

One such model is the Efficient Architecture for Running Threads (EARTH) designed in the late 90s. EARTH's goal was to provide a simple, efficient, and evolutionary execution model to enabling the construction of a full scale multiprocessor utilizing commodity hardware. The following is a summarization of work done by Theobald [45], Hendren [28], and Hum [29].

2.2.1 EARTH Threading Model

EARTH attempts to take advantage of many ideas set forth by dataflow while addressing issues such as the lack of locality found in a DFG. The EARTH thread model is hierarchical containing two levels of threading, fibers and threaded procedures.

2.2.1.1 Fibers

Fibers are asynchronous, non-preemptive, sequences of instructions. Each fiber has a synchronization slot which determines when a fiber fires. A synchronization slot contains a counter and the number of dependencies needed to fire a fiber. A fiber is executed based on the following firing rule:

Fiber Firing Rule: A synchronization counter is set to the number of dependencies required. Once the counter reaches zero, all dependencies have been satisfied enabling the fiber. The fiber may fire once a processing element is available.

Fibers are connected through their data dependencies forming a graph similar to a DFG. Parallel execution of fibers is permitted given each fiber is enabled and processors are available.

2.2.1.2 Threaded Procedures

Threaded procedures combine ideas from both explicit token storage and argument fetching dataflow. A threaded procedure contains at least one fiber, the local variables shared between fibers, and parameters passed to the procedure. Fibers and threaded procedures are tightly coupled, as each fiber must belong to a single threaded procedure. Threaded procedures are invoked by running fibers (except for an initial procedure). Every threaded procedure has at least one fiber, called fiber 0, which is the first in the procedure to fire. Fibers contained within the threaded procedure may produce data and signals for fibers both contained within and outside their own threaded procedure. A threaded procedure is maintained in memory until a fiber terminates the procedure, causing its deallocation including the procedure's fibers and data. The EARTH model takes advantage of two levels of parallelism; threaded procedures execute in parallel as well as the fibers they contain.

2.2.2 EARTH Architecture Model

Since EARTH has played an important role in developing the Codelet model, it is useful to look beyond its threading model to its architecture model. The EARTH architecture model demonstrates the structure and organization of an abstract machine capable of supporting the threading model. An EARTH system is comprised of at least one EARTH node connected by some network. An EARTH node is made up of the following components:

- Execution Unit (EU) - May contain one or more processing elements (cores or threading units) and is responsible for executing fibers
- Synchronization Unit (SU) - A processing element responsible for synchronizing fibers and threaded procedures as well as handling remote data accesses.

- Event Queue (EQ) - A queue containing operations produced by the EU for the SU to execute.
- Ready Queue (RQ) - A queue of fibers which are ready to be executed

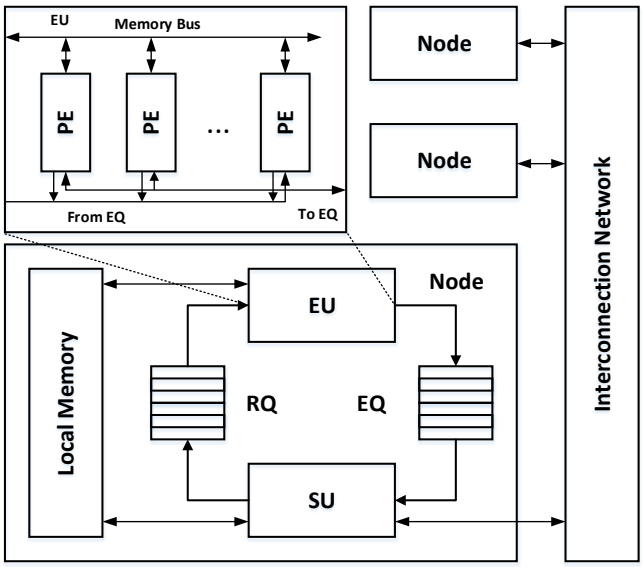


Figure 2.2: The EARTH architecture model describes physical components and their configuration required to run the threading model.

An EARTH node's EU and SU communicate through two queues, the EQ and RQ. As fibers are executed in the EU, they produce synchronizing instructions (e.g. satisfying another fiber's dependency). The SU is responsible for decrementing the synchronization slot of other fibers and writing remote data. This separation of task was intended to address the long latency and limited bandwidth of commodity processors available in the 90's. By decoupling the synchronization from execution, the EU is not required to suspend execution. Traditionally, this problem is handled through preemption; however this requires either register windows or pushing and pulling execution contexts to and from slower memory.

2.2.3 EARTH's Advantages

The EARTH model was designed to explore dataflow based multiprocessing in an evolutionary manner via off-the-shelf hardware available at its inception. Despite being designed before the commonplace of multi- and many-core architectures, its creators provided several innovations meaningful for today and future architectures.

An important advancement is the hierarchical expression of parallelism provided by the programmer. Relying on dataflow semantics, both fibers and threaded procedures may run in parallel. By grouping fibers together, programmers are capable of expressing both a fine-grained parallelism encompassed in a more familiar function-like wrapper. While, fibers are intended to run in parallel, an EU was typically a single processor (at the inception of the EARTH model). EARTH's architecture model permitted the exploration of multi-threaded execution without actual multi-core technology. With the advent of multi- and many-core, hierarchical threading has much more significance.

Another important development is EARTH's balance between parallelism and locality. While EARTH explores dataflow inspired parallel execution, it does so at a macro-dataflow level, providing the exploitation of temporal and spatial locality found in traditional von Neumann architectures. Moreover, the grouping of fibers in threaded procedures provides a means for sharing data leading to more locality.

The Codelet model builds on these contributions, further discussing them in many-core context in following chapter.

Chapter 3

CODELET MODEL

The previous chapter describes several parallel execution models, all developed before the multi- and many-core era. These models primarily focus on the exploitation of fine-grain parallelism. As we look forward to exa-scale, we are forced to address the pressing energy constraints imposed by the physical world. In doing so, parallelism must effectively be balanced with locality and tailored to the large scale heterogeneous machines of tomorrow. A philosophical document has outlined our initial views of a Codelet model [25]. The following chapter proposes an updated specification reflecting our experience working with several codelet based runtimes. The Codelet model is a hybrid von Neumann/dataflow execution model [50, 44]. The model is a means to develop the underlying system architecture and system software for exa-scale computing. Based on a configurable abstract machine model, the Codelet model extends fine-grain multi-threading to provide facilities to address resource utilization including locality. The current chapter discusses the abstract machine model in section 3.1. Section 3.2 presents the threading model followed by the Codelet architecture model presented in section 3.3.

3.1 Codelet Abstract Machine Model

In conceiving an execution model for exa-scale, it is imperative to consider the characteristics of current and future many-core systems. For that reason we present the abstract machine model, a representation of the type of system well suited for the Codelet model. The figure 3.1 depicts our view of future many-core systems.

The codelet abstract machine model consists of many nodes connected together via some interconnection network. Contained within a single node are multiple chips,

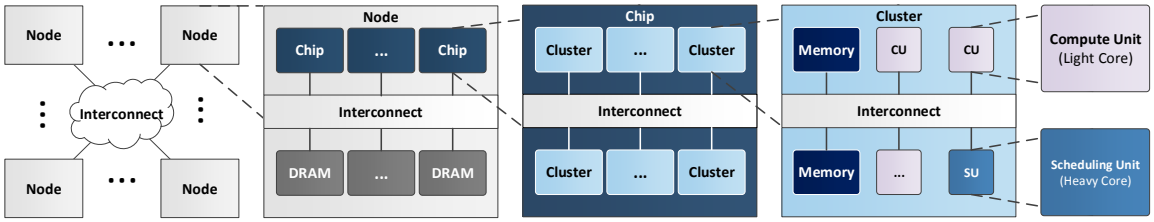


Figure 3.1: The codelet abstract machine is hierarchical and heterogeneous. Moreover, the machine will be extremely parallel and with various levels of memory (each with different latencies).

each with DRAM memory coupled by some interconnect. Within a chip exists groups of heterogeneous cores with both shared memory and local memory, called clusters. We envision two types of cores:

- Computation unit - A core specialized for executing codelets (e.g. sufficient for executing general purpose code). This core is connected with other computation units within its cluster, but lacks access to remote memory or IOs.
- Synchronization unit - A core responsible for servicing events (signaling codelets) and load balancing codelets and TPs. This core has access to remote memory and IOs.

We believe that this high level view is specific enough to provide a solid foundation for the Codelet model. In summary a codelet capable system will have the following properties:

- Parallel - These systems will have plenty of compute resources available.
- Hierarchical - The system must be hierarchical to effectively manage the compute resources available to the programmer.
- NUMA - Interconnects binding components at several levels will have varying latencies.
- Heterogeneous - Different cores will be specialized for different tasks like computation intensive cores and synchronization cores.

3.2 Threading Model

The threading model details how parallel programs are expressed in a parallel system. The Codelet model employs hierarchical, fine-grained, event-driven multi-threading. The model is designed to explore parallelism to sufficiently utilize a large scale systems, while still providing mechanism to ensure locality. Striking the right balance between parallelism and locality is paramount in achieving an energy efficient system.

3.2.1 Codelet

The Codelet execution model is centered on the concept of a codelet. A codelet is a sequence of machine instructions. A codelet is the principal scheduling quantum in the model as it is scheduled atomically. In order to better utilize the underlying system, codelets are event-driven. Each codelet has an event synchronization slot which represents the dependencies required by the codelet before it may execute. The most common event is the availability of data (i.e. satisfaction of a data dependency); however it is not limited to this alone. Events include the following:

- Data dependencies - A codelet may require specific data produced by another codelet
- Control dependencies - A codelet may signal another codelet based on some condition similarly to a dataflow control actor [19]. Doing so may change the execution path of a CDG enabling certain codelets while disregarding others.
- Locality dependencies - A codelet may prefetch data for other codelets to use concurrently. This differs from the data dependence since the event is used to indicate the location of the data in memory.
- Energy events - Different functionally equivalent codelets may have different performance versus energy trade-offs. Energy events could decide which codelets to execute.
- Other resource requirements (e.g. bandwidth, network, etc.)

A codelet is executed, or fired, based on the following rule:

Codelet Firing Rule: Each codelet's synchronization slot is used to keep track of the events required prior to codelet execution. When all of a codelet's event have occurred,

the codelet becomes enabled. Once a processing element (core) is available, the codelet may be fired.

As each codelet is executed, it satisfies events required by other codelets. A codelet's output is not atomic, meaning it may satisfy an event and continue running. This is contrary to traditional dataflow semantics which complete an actor's execution before signaling. This provides a more natural overlapping of codelet capable of executing in parallel. Enabled codelets may execute in parallel since all of their events (i.e. dependencies) have been satisfied.

Codelets are very similar to EARTH's fibers. The main difference being codelets are event driven versus data driven fibers. This distinction is important for future architectures as events generated from an intelligent runtime or hardware units may shape the execution of an application. By generalizing signaling, we provide the facilities for future work in areas such as energy management and resiliency.

3.2.2 Codelet Graph

Connecting codelets and their dependencies via events naturally form a graph similar to a dataflow graph. This interweaving of codelets is called a CoDelet Graph (CDG), and is useful for reasoning about a given application. A CDG is a directed graph consisting of three components:

Codelet Represented by a node, indicate a sequence operations to be performed.

Event Represented by an arc. The arcs entering or exiting a node indicate a potential signal from a codelet representing an event.

Token Represented by a point on an arc indicating a signal.

The CDG is a very effective mechanism to observe the available parallelism in a given application. By studying the CDG, one can locate regions of an application containing both plentiful and sparse amounts of parallelism.

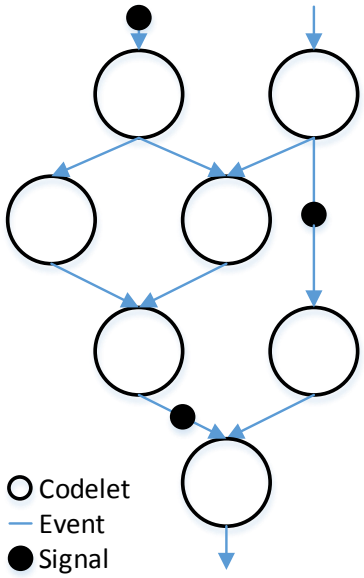


Figure 3.2: A CoDelet graph (CDG) contains codelets, events, and tokens.

3.2.3 The Ideal Codelet

The codelet model proposes a method of expressing programs foreign to many programmers. Furthermore, there are (efficient) solutions to some challenges presented by applications which would not be possible in a dataflow system. To not exclude these solutions as they could present a more optimal approach, the Codelet model permits certain caveats to the ideals of the model at the programmer's risk. The following presents the "ideal codelet." Breaking these ideals does not necessarily imply an incorrect program, but it may negate certain properties such as determinate execution.

3.2.3.1 Non-Preemptive

Similar to the EARTH model, codelets should be not be preemptable. Today's traditional multi-threaded systems may suspend execution of a thread due to some external event. This requires the context of the thread including the thread's frame and registers be swapped with another thread. This is costly in both time and energy,

as moving data is expensive. Codelets should be fine-grained enough to permit it to run to completion.

Moreover, some systems use the same methodology to tolerate long latency operations executed by a thread. This "voluntary yielding" is not appropriate for the Codelet model as codelets should have all data and resources required locally before firing. In ensuring the availability of resources before execution, a codelet will not need to be suspended as the operations will have short latencies. However, an architectures with features like register windows might not suffer the same costs in time and energy switching between windows. In this type of system, yielding might be a reasonable concession.

Another technique capable of mitigating effects of long latency operations is to use split-phase transactions. A split-phase transaction divides a single codelet containing a long latency operation. The first codelet ends after initiating the operations. One or more codelets will continue after it is complete (and the continuing codelets have been signaled). Dividing the codelet permits the computation unit an opportunity to fire other enabled codelets while the long latency operation is still outstanding.

3.2.3.2 Well-Behaved CDGs

The dataflow model can provides strong guarantees like determinate and repeatable execution [20]. These properties emanate from well-behaved schemata [19] and side-effect free actors.

An actor is well-behaved if and only if an actor consumes all of its input tokens (one for each input arc), and produces an output token for each output arc. An acyclic, simple dataflow graph (e.g. a graph without control gates or merge nodes) is determinate [19, 37]. Assuming a codelet is side-effect free, a codelet is similarly well-behaved if and only if each possible input *data event* occurs, and the codelet fires signaling all of its output *data events*. Moreover, an acyclic combination of these determinate codelets form a determinate CDG.

3.2.3.3 Well-Formed CDGs

Traditional dataflow contains non-well-behaved actors used to implement control-flow. These actors include a switch, merge, and true and false gates. Using the construction rules provided in [19], these actors can be used to create well-formed graphs. A well-formed graph is also a well-behaved graph, however the contrary is not always true.

Argument-fetching dataflow modifies the dataflow graphs to a signal graph by separating data from the signaling of actors. Signal graphs do not fire switch or merge actors. Rather, the same functionality is achieved by properly linking the nodes in a graph and conditionally signaling consuming nodes as shown in [14]. The Codelet model follows this approach in implementing conditional statements.

More recently, a homogeneous form of dataflow has demonstrated the same properties exhibited by traditional dataflow [47]. This approach requires static dataflow semantics. That is, an actor must not be capable of firing until its output links are free. The actors proposed are analogous to the nodes in the signal graph and also codelets in a CDG. To leverage the proof from homogeneous dataflow, the Codelet model must also observe static dataflow's single-token-per-arc rule. Multiple tokens on an arc originate from either loops, recursive function invocations, or streaming data.

In order to support recursive function invocations and parallel for loops, a CDG is duplicated (using Threaded Procedures proposed in section 3.2.4) and linked appropriately avoiding multiple tokens per arch. Current research is being conducted to determine how best to support streams.

3.2.4 Threaded Procedure

Similar to the EARTH model, asynchronous functions are called Threaded Procedures (TPs) in the Codelet model. Much like its ancestor, the codelet model's TPs are containers for a CDG. A TP also features a frame which holds the inputs passed to it, the resulting output, and data shared by the contained codelets. TPs are invoked (like a function call), and exist in memory until all of the codelets in the CDG have

finished executing. By grouping all of the codelets and necessary data together, a TP provides a convenient mechanism for codelets to access data and other codelets (important for signaling events). A TP and its codelets are confined to a single cluster and residing in cluster memory ensuring locality.

3.2.5 Loop

A loop in the codelet model is represented by a cycle in a CDG. That is the loop body is comprised of several codelets. There are two forms of loop parallelism exploited in the codelet execution model. The first and simplest is *parallel for* loops. These kinds of loop do not have any loop-carried dependencies permitting the iterations to run in parallel. The second form of loop parallelism is codelet software pipelining. Current research is being conducted to determine how to utilize software pipeline for the Codelet model [49].

3.3 Codelet Architecture Model

The previous sections detailed the abstract machine model and the threading model. The following describes how the threading model is mapped to the abstract machine model. The Codelet architecture model describes the responsibility of the components that comprise a system capable of exploiting the Codelet model.

The abstract machine outlines a hierarchical system including nodes, clusters, synchronization units, and computation units. The Codelet architecture model includes three queues in addition to the component in the abstract machine model. These components are as follows:

- Synchronization Unit (SU)
- Computation Unit (CU)
- Threaded Procedure Pool (TPP)
- Codelet Ready Pool (RP)
- Event Pool (EP)

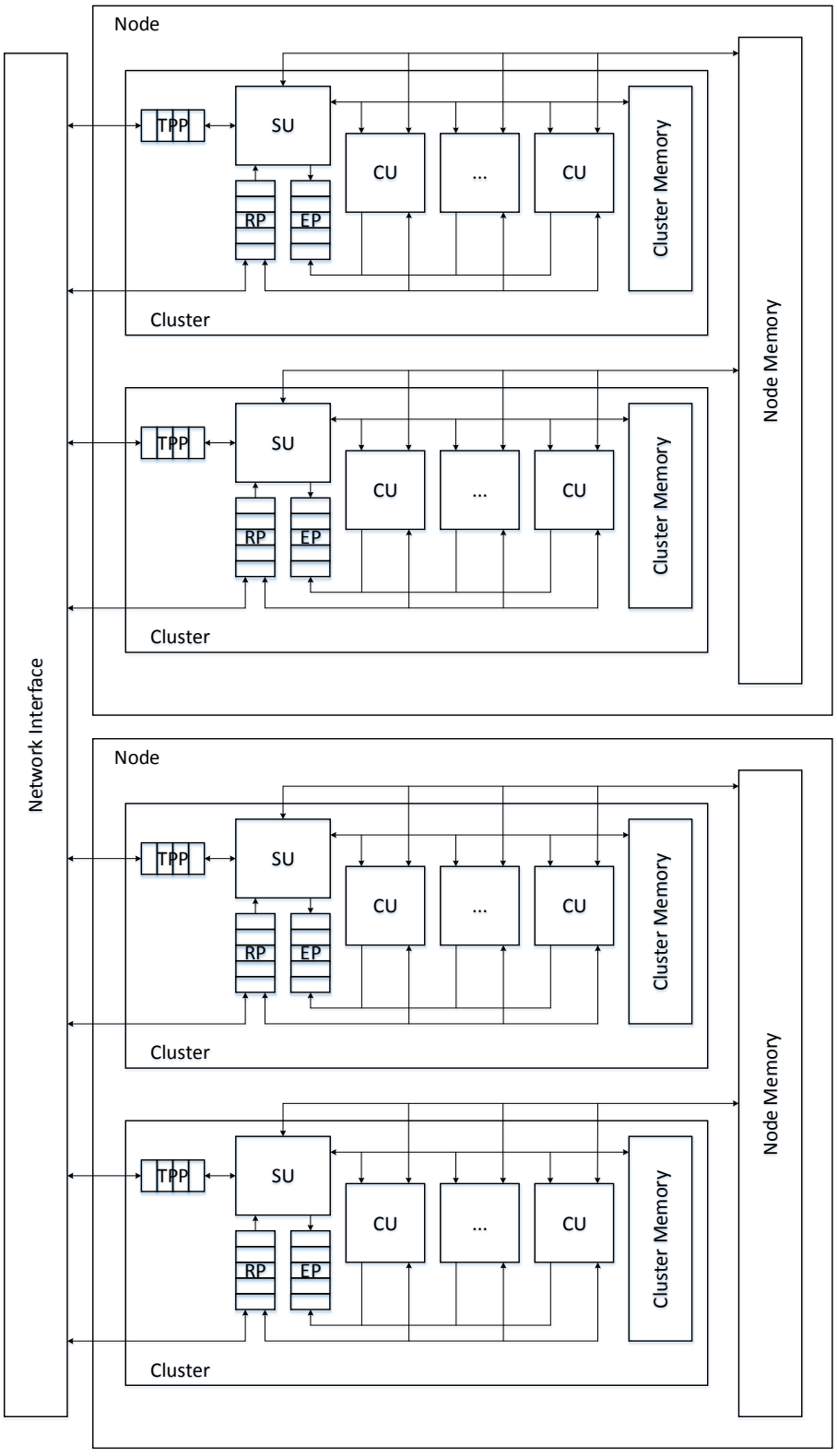


Figure 3.3: Codelet architecture model depicting two nodes with two clusters each.

3.3.1 Synchronization Unit

The synchronization unit has several responsibilities including the distribution and synchronizing of the execution of codelets across clusters and nodes. The SU's primary responsibility is to load balance TPs both within and outside a node. TPs are claimed by a SU via the TPP. Once acquired, a TP's codelets are bound to the cluster. The TP's context (codelets and frame) are pushed into cluster memory, and the enabled codelets are pushed to the RP. Lastly, the SU synchronizes codelets outside of the cluster by servicing requests from the EP.

A SU will claim a TP based on the load of codelets already executing on its cluster. If the cluster has no codelets running, the SU will try to claim a TP from its own TPP. If there are no TPs available in the TPP, the SU will attempt to claim TPs from neighboring clusters in a work stealing fashion. A SU may try to buffer codelets by claiming TPs while other codelets are still running on its cluster. This is considered part of the SU's scheduling policy and is part of a particular codelet implementation. One example of a SU's scheduling policy is to try to maintain a certain amount of codelets ready to run in the RP.

3.3.2 Computation Unit

The computation units are responsible for executing codelets which make up an application. Several CUs are grouped together to form a cluster. A CU has access to all of the memory contained in the node including cluster and node memory. Codelets are acquired from the RP by the CUs. CUs are responsible for directly signaling codelets which are local to the cluster. Once a codelet is enabled, it is placed in the RP where it waits for an available CU. If a codelet resides outside its cluster, the executing CU generates an event request and places it in the EP. When an enabled codelet is fired by a CU, its execution continues until the codelet is complete (its execution is not moved between CUs).

3.3.3 Pools

There are three pools in the Codelet architecture model which serve as the glue between the event and codelet executing components, the Threaded Procedure Pool (TPP), Ready Pool (RP), and the Event Pool (EP). Each cluster contains these three pools.

The TPP holds the threaded procedures available which have not begun execution. When a running codelet invokes a TP, the TP is immediately placed into the TPP. SUs are responsible for claiming TPs from the TPP based on the SU's scheduling policy.

The RP contains all of the enabled codelets which are ready to run in a cluster. The RP is accessed by both the SU and the CUs within a cluster. When a TP is claimed, the SU will place the enabled codelets into the RP beginning the TP's execution. The codelets placed in the RP by the SU are codelets that do not require any event to fire. Codelet that are not enabled exist in memory local to the cluster. Once these codelets become enabled, they are placed into the correct RP by the CU or SU responsible for enabling them.

Signals that satisfy an event outside of a codelet's cluster generate an event request, which is placed in the EP. The SU accesses the EP, and satisfies the event's request potentially enabling codelets outside of the cluster.

3.3.4 Memory and Interconnect

A complete memory model is comprised of an addressing mode and memory consistency model. A memory model's purpose is to provide an agreement on the semantics of memory operations between hardware and software to ensure proper execution of the user's program.

The previously mentioned philosophical document proposes a codelet architecture with a Location Consistency (LC) based memory consistency model [25]. LC is a sufficiently weak model which respects causality. In LC, the content of a memory location is not a monolithic value, but rather a partially ordered multi-set (POMSET).

Operations (read, write, acquire, and release) to the POMSET are defined by Gao and Sarkar [24]. A memory consistency model is enforced (primarily) by physical hardware. Memory consistency models like sequential consistency permit a narrow view of memory, forcing hardware to synchronize (potentially unnecessarily). This synchronization is not only costly in time, but also energy. This motivates our recommendation for LC as a strong candidate for future systems.

Another possible scenario is for codelets to leverage a write-once policy. Under this policy, codelets would not "release" produced data until their completion. This idea is explored further in [17].

The appropriate addressing mode of a codelet based system is still under investigation. One potential idea is the use of Globally Unique Identifiers (GUIDS). GUIDS would be assigned to codelets and TPs providing a unique name (address) permitting access across distinct memories (i.e. distribute memory systems).

The codelet threading model only requires an atomic decrement used to decrease the synchronization slot. Additional atomic operations, like the popular compare-and-swap are not required, but helpful in designing efficient data structures used in a codelet runtime. Furthermore, the Codelet architecture model does not require a particular topology for the interconnection network. By limiting the demands of the Codelet model, it may be implemented on a wider variety of systems.

Chapter 4

DARTS

While an execution model is implemented via an entire system, a runtime can augment hardware not designed for a particular model. The Delaware RunTime System (DARTS) is a runtime written for shared memory x86 architectures to implement the Codelet execution model. While there are other codelet based runtimes [32, 15], DARTS attempts to remain as accurate as possible to the aforementioned model with the intent of analyzing and further developing codelets. DARTS is written in C++, and takes advantage of object oriented programming. Using C++ is low level enough to provide sufficient control over the underlying hardware while still promoting modularity and portability. The following is the description of the DARTS implementation and an evaluation using micro benchmarks.

4.1 Implementation

The codelet threading model is comprised of three elements, codelets, TPs, and loops, which are implemented as C++ objects. Each object uses inheritance and virtual functions to define a particular instance of the type of object.

4.1.1 Codelets

Codelets are the core of a DARTS application, and are derived from a base class called Codelet. The base codelet class has several important fields including a synchronization slot, reference to a TP, and some space for metadata. The synchronization slot stores the total number of events required to fire, and a counter. The counter is set to the requisite number of events and is decremented until it reaches zero which enables the codelet. Once a codelet is enabled, it is ready to execute using the virtual

fire method which contains the codelet code. The fire method is a pure virtual method defined in the base codelet class and implemented in a inherited class specialized by the application programmer. While the fire method is executing, it has access to the codelet's TP reference enabling it to access data and other codelets in the TP. Moreover, the metadata associated with a codelet permits additional information useful for scheduling as demonstrated in section 4.2.1.4.1.

A codelet has four input parameters which directly correspond to its members. A specialized codelet must provide the number of requisite events, a reset event count, a reference to its TP, and metadata to the base classes constructor. A codelet may access its TP reference via the `myTP_` member. The following is an example codelet.

```
1  #include <iostream>
2  #include ``darts.h''
3  using namespace darts;
4
5  class SomeCodelet : public Codelet
6  {
7  public:
8    SomeCodelet (uint32_t EventCount, uint32_t EventReset,
       ThreadedProcedure * TPReference, uint32_t Metadata):
9    Codelet(EventCount, EventReset, TPreference, Metadata) { }
10
11   virtual void fire(void)
12   {
13     std::cout << ``I am a codelet from '' << myTP_ << ``!'' << std::
       endl;
14   }
15 };
```

4.1.2 Threaded Procedures

TPs are designed to be containers for codelets and shared data. As such, a TP inherits from a base class called ThreadedProcedure, and is extended with data

and codelets. A typical TP contains references to input data, internally shared data (between codelets), output, codelets, and a reference to codelets to signal outside of the TP. Codelets using their TP reference are capable of writing and reading shared data. During the TP's construction any enabled codelets should be passed using the add method. The add method takes a reference to a codelet and passes the codelet to the runtime to begin its execution. The following is an example TP.

```
#include <iostream>
#include ``darts.h''
using namespace darts;

class SomeTP : public ThreadedProcedure
{
public:
  int x, y;
  int * result;
  CodeletOne cd1;
  CodeletTwo cd2;
  Codelet * toSignal;

  SomeTP(int * Result, Codelet * ToSignal):
    ThreadedProcedure(),
    result(Result),
    toSignal(ToSignal),
    cd1(0, 0, this, 0),
    cd2(2, 2, this, 0),
    toSignal(ToSignal)
  {
    add(&cd1);
  }
};
```

TPs and all of their members are allocated on the heap. DARTS employs a reference count based garbage collection to automatically delete TPs when they are no

longer needed. A simple counter is used to keep track of the number of outstanding enabled codelets and children TPs (TPs that have been invoked by contained codelets). This ensures that all of a TP's executing codelets have access to the data residing in the TP as well as codelets from children TPs. Once a TP's reference count reaches zero, it will decrement its parent TP, and the TP will be deleted. It is necessary to include children TPs in the reference count since a TP may have invoked its child TP performing a split-phase transaction. During this time it is possible there will be no codelets running from the parent TP. In this case the parent TP would be deleted leading to a fault.

This method of garbage collection is sufficient for most codelet applications, however not for all. If an application requires data to persist past the execution of its codelets, the data, if stored in the TP, should not be deleted until specified by the programmer. For this reason, methods are available to the programmer to artificially increase and decrease the reference count ensuring the TP will remain in memory for as long as desired.

4.1.3 Loops

DARTS implements three types of loops, a serial loop, a codelet for all loop, and a TP for all loop. These loops give the programmer the flexibility to decide to some degree where parallel iterations will be executed, whether it be on a single core, single cluster, or the entire system. This allows the programmer to explore the trade-off parallelism versus locality.

Loops in DARTS are implemented in two parts, a specialized loop class and a control codelet. Specialized loop classes inherit from a base class called Loop. The base Loop class has two members, an iteration id and codelet to signal when the loop is finished. The specialized Loop class is comprised of a CDG and the data shared by the contained codelets. Codelets without any event requirements are passed to the runtime during the loops construction identically as TP's enabled codelets. The following is an example of the loop base class.

```
#include <iostream>
#include ``darts.h''
using namespace darts;
class SpecialLoop : public Loop
{
public:
  int x, y;
  CodeletOne cd1;
  CodeletTwo cd2;

  SpecialLoop(unsigned int Id, Codelet * ToSignal):
    loop(Id, ToSignal),
    cd1(0, 0, this, 0),
    cd2(1, 1, this, 0)
  {
    add(&cd1);
  }
};
```

The second part of a loop in DARTS is the loop controlling codelet. Leveraging C++'s templating, we implement three controlling codelets, the serial loop, the codelet for all loop, and the parallel for all loop.

4.1.3.1 Serial loop

A serial loop will loop over a CDG for a given number of iterations sequentially in a single cluster. This is not to say that all the codelets in the CDG are executed sequentially, rather that each iteration of the CDG is executed in order. A serial loop takes a derived loop type as its first template parameter, and allocates a single copy of the CDG on the heap. The serial loop object is implemented as a special codelet which is responsible for starting and stopping the loop. When this special codelet is constructed (during TP construction time) the loop CDG is allocated and the iteration id is set to zero. Once the serial loop codelet's synchronization counter

reaches zero, the loop body begins execution. When an iteration is completed, the serial loop codelet is again signaled (transparently to the programmer). If there are still iterations remaining to be executed, the loop's iteration id is incremented, and the loop is reconstructed using C++'s in place allocator permitting the next iteration to begin. When all iterations are completed, the serial loop codelet signals its completion to the subsequent codelet provided at construction time.

The serial loop codelet takes at least three parameters:

- A number of requisite events to start the loop
- The number of events (signals from codelets contained in the loop) to end an iteration
- A codelet to signal once the loop is completed
- Number of iterations to perform

The CDG contained in the loop body will most likely require additional data beyond the loop construct's arguments. To pass this data into the loop's CDG, the data (as parameters) are appended onto the loop class' constructor method. The data can then be passed to the serial loop codelet's constructor (after the required parameters) in the order they appear in the loop constructor. The types of the additional inputs can be inferred by most modern compilers. Alternatively, one can pass the types of the additional inputs as template parameters after the loop type to the serial codelet. The inputs to the loop are stored and provided by the runtime during the loop's construction. As such, they are fixed and cannot be changed after the serial loop codelet's construction. If a more dynamic approach is required, one can pass a reference to a variable rather than a static value. This is useful if data required by the loop body is not known when the loop controlling codelet is created. The following shows an example of a serial loop. The loop CDG requires an addition integer, Number. Notice how this affects the constructors of the loop and serial for.

```
1 #include <iostream>
2 #include ``darts.h''
3 using namespace darts;
```

```
4
5  class FormalLoop : public Loop
6  {
7  public:
8    int number;
9    HelloCodelet hello;
10   GoodByeCodelet goodbye;
11
12   FormalLoop(unsigned int Id, Codelet * ToSignal, int Number):
13     loop(Id, ToSignal),
14     number(Number),
15     hello(0, 0, this, 0),
16     goodbye(1, 1, this, 0)
17   {
18     add(&hi);
19   }
20 };
21
22 class TPwithSerialLoop : public ThreadedProcedure
23 {
24 public:
25   serialFor<FormalLoop> serialLoop;
26
27   TPwithSerialLoop(Codelet * ToSigal, int Iterations, int Number):
28     serialLoop(0, 0, this, 0, ToSignal, Iterations, Number),
29   {
30     add(&serialFor);
31   }
32 };
```

4.1.3.2 Codelet For All Loop

A codelet for all loop differs from a serial codelet loop as its iterations may execute in parallel within a single cluster. The codelet loop has a nearly identical

form to the serial loop (except the name of the object) requiring the same parameters, template parameters, and parameter order. The differences between the codelet loop and the serial loop are evident during their respective construction (which is invisible to the application programmer). The codelet loop codelet allocates a separate loop CDG for each iteration upon creation. Each iteration is constructed with a unique iteration id (ranging from zero to the total number of iterations minus one) and a reference to the codelet loop codelet. The arguments required by the loop CDGs are passed by the runtime to the loop constructor. The codelets from all of the iterations may run in parallel within the same cluster. As each iteration finishes, the codelet loop codelet is signaled. Once all of the iterations are complete, the codelet loop codelet signals the next codelet provided during construction. The following is an example of a codelet for all loop.

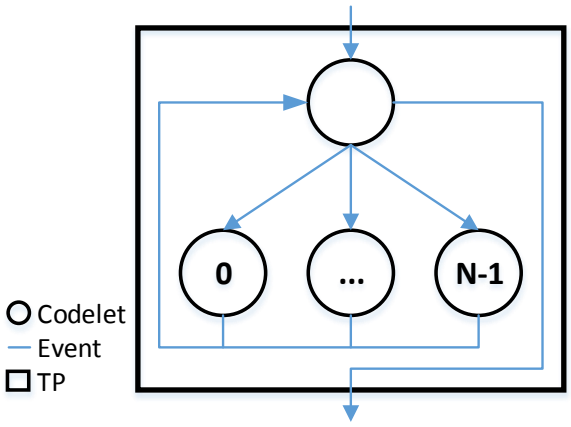


Figure 4.1: A codelet for loop spawns N loop iterations (one codelet per iteration in this figure). These iterations are all executed in a single cluster. Upon completion, the iterations signal the codelet that initiated the loops which continues execution.

```
1 #include <iostream>
```

```
2  #include ``darts.h''
3  using namespace darts;
4
5  class SemiInformalLoop : public Loop
6  {
7  public:
8    int number;
9    HiCodelet hi;
10   ByeCodelet bye;
11
12   SemiInformalLoop(unsigned int Id, Codelet * ToSignal, int Number):
13     loop(Id, ToSignal),
14     number(Number),
15     hi(0, 0, this, 0),
16     bye(1, 1, this, 0)
17   {
18       add(&hi);
19   }
20 };
21
22 class TPWithCodeletLoop : public ThreadedProcedure
23 {
24 public:
25   codeletFor<SemiInformalLoop> codeletLoop;
26
27   TPWithCodeletLoop(Codelet * ToSigal, int Iterations, int Number):
28     codeletLoop(0, 0, this, 0, ToSignal, Iterations, Number),
29   {
30     add(&codeletLoop);
31   }
32 };
```

4.1.3.3 Parallel For All Loop

The parallel for all loop follows the same form as the previous two loops, however its iterations are permitted to be executed in parallel on multiple clusters. The parallel loop treats each iteration of the loop as a TP permitting the iterations to be load balanced across clusters. During the parallel loop codelet's construction, no loop CDGs are allocated. Instead the loop arguments are stored, and once the parallel loop codelet fires, a TP is invoked with the loop CDG for each iteration. The TPs are released into the runtime, load balanced, and executed. Once the iterations are complete, the parallel loop codelet is again fired which signals the next codelet to run. The runtime is responsible for deleting the TPs as discussed in section 4.1.2. The following is an example of a parallel for loop.

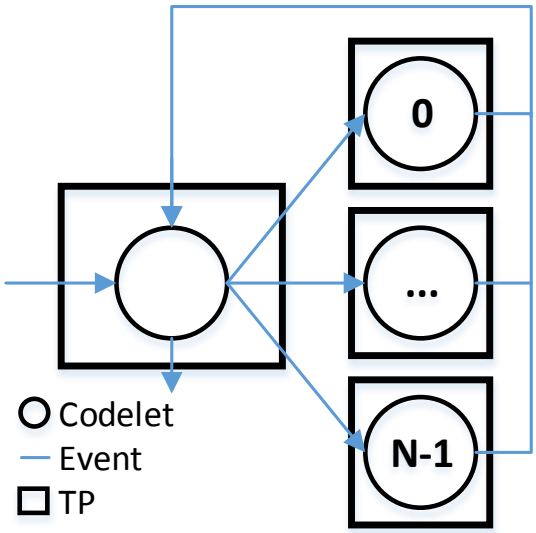


Figure 4.2: A parallel for loop spawns N parallel loop iterations as stand alone TPs. These TPs can run on multiple clusters. Upon completion, the iterations signal the codelet that initiated the loops which continues execution.

```
#include <iostream>
#include ``darts.h''
using namespace darts;

class InformalLoop : public Loop
{
public:
  int number;
  HeyCodelet hey;
  PeaceCodelet peace;

  InformalLoop(unsigned int Id, Codelet * ToSignal, int Number):
    loop(Id, ToSignal),
    number(Number),
    hey(0, 0, this, 0),
    peace(1, 1, this, 0)
  {
      add(&hey);
  }
};

class TPWithParallelLoop : public ThreadedProcedure
{
public:
  codeletFor<InformalLoop> parallelLoop;

  TPWithParallelLoop(Codelet * ToSigal, int Iterations, int Number):
    parallelLoop(0, 0, this, 0, ToSignal, Iterations, Number),
  {
    add(&parallelLoop);
  }
};
```

4.2 Runtime

The DARTS runtime is designed to fill in the gaps between the hardware and the Codelet architecture model. The runtime itself is design as a single class with two primary responsibilities. The first is to use an inputted system configuration to initialize the cores for execution. Since not all systems are heterogeneous, the runtime with programmer guidance will elect some cores to be synchronization units and others to be computation units. The second responsibility of the runtime is to take the initial TP and begin execution. The runtime is comprised of four types of objects, Threaded Procedure (TP) schedulers, CoDelet (CD) schedulers, an abstract machine configuration, and a final codelet.

4.2.1 Schedulers

The most vital components of the DARTS runtime are its schedulers. Every core in the system has at least one scheduler (hyper-threaded cores may have more). Schedulers are built on top of POSIX threads which are pinned to a single core. The schedulers have varying responsibilities including scheduling codelets, scheduling TPs, satisfying codelet's events, and firing the codelets. The mechanisms and methods a scheduler utilizes to performs these task are considered its policy. There are two types of schedulers which directly correlate to the codelet architecture model.

- Threaded Procedures (TP) schedulers correspond to the synchronization unit, ready pool, and threaded procedure pool.
- CoDelet (CD) schedulers correspond to the computation unit and the ready pool.

4.2.1.1 TP Scheduler

TP schedulers perform the responsibilities of the synchronization unit presented in section 3.1. In addition, each TP scheduler contains a data structures to hold TPs and codelets which are ready to run. Thus the TP scheduler implements the SU, RP, and TPP in a single class along with appropriate methods to access each component.

Several computation units are joined with a single TP scheduler to form a cluster. The TP scheduler's primary responsibility is to load balance the system both inside

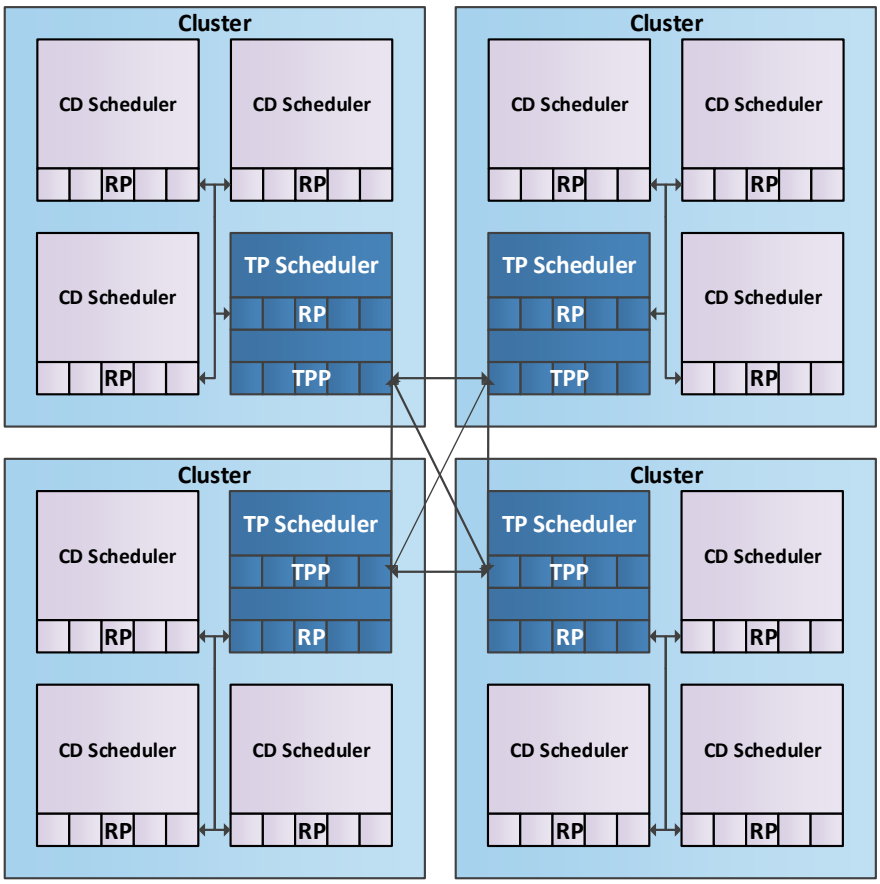


Figure 4.3: DARTS is comprised of Threaded Procedures (TP) and CoDelet (CD) schedulers. A TP scheduler has a Ready Pool (RP) for enabled codelets, and a TP Pool (TPP) for TPs which have not begun execution. A CD schedule contains a local RP.

and outside its cluster ensuring work is efficiently distributed. This job is accomplished utilizing two methods each matching the two levels of threading found in the Codelet model. The first is to load balance TPs between other TP schedulers requiring a TP scheduler to have access to all of the TP schedulers in the runtime. When a new TP is invoked, it is placed in the TP scheduler's TPP. When the cluster has little or no codelets enabled, the TP scheduler will take an available TP from its TPP. If there are no TPs available, the TP scheduler will "steal" a TP from another TP scheduler's TPP. This way TPs and their codelets are distributed across clusters. As previously

mentioned, particular details about when and how TP schedulers steal TPs in addition to the data structure used to implement the TPP make up the TP scheduler's policy which is described in section 4.2.1.4.

The second method by which a TP scheduler balances work is by distributing codelets to computation units. When a TP scheduler claims a TP from its TPP, it takes the TP's enabled codelets and places them in the RP. Using the RP, the TP scheduler distributes codelets to the computation units in its cluster. Moreover, as more codelets become enabled, they will be placed in the RP requiring the TP scheduler to distribute them. The TP scheduler can distribute codelets employing several approaches. For example, the TP scheduler may distribute codelets in a round robin fashion cycling through all of the computation units giving each one. In this fashion, the TP scheduler is also responsible for load balancing codelets within the cluster.

In the codelet architecture model, the SU is only responsible for load balancing and satisfying events from the EP. Since DARTS is implemented for a shared memory system, each computation unit is capable of satisfying events (i.e. decrement a synchronization slot) independently. This frees the TP scheduler to only have to load balance the system. Depending on the TP scheduler's policy and the application, the TP scheduler can become underutilized. To mitigate this inefficiency, TP schedulers in DARTS are permitted to run codelets. This creates an interesting balance between executing codelets and load balancing the rest of the system.

4.2.1.2 CD Scheduler

The CD scheduler maps to the codelet machine model's computation unit. Several CD schedulers are bound together with a single TP scheduler to form a cluster. Each CD scheduler is implemented as a class with a pool of local codelets to run. This can be seen as an extension to the RP (the RP is a distributed data structure in DARTS). This pool is filled by the TP scheduler or by the CD scheduler depending on the policies chosen for each. The CD scheduler's role is to execute enabled codelets. The CD scheduler will execute a codelet by running a codelet's virtual fire method. If

a codelet invokes a new TP, the CD scheduler will place the TP into the TP scheduler's TPP. Moreover, if the executing codelet signals an event, the CD scheduler will decrement the appropriate synchronization slot. If a codelet becomes enabled, the last CD scheduler to signal an event will place it in the enabled codelet's cluster TPP.

Allowing codelets to signal any codelet in an application is a departure from the codelet machine model presented in section 3.1. This optimization is only possible in a shared memory system, where all codelet instances are visible to all processors. In a distributed system, one would need to locate the codelet instance before signaling it, implying a long latency operation which is better suited for the SU.

4.2.1.3 Abstract Machine

DARTS utilizes the abstract machine presented in section 3.1 to provide a way to configure the runtime. The abstract machine presents a view of the system which divides nodes into clusters containing several CUs and one SU. In addition, each cluster has some local memory used to store running TPs. Most x86 systems today are cache based with multiple level of sharing. For example, a single node may have a private L1 cache for each core, but shared L2 and L3 caches. In a homogeneous system, this provides several options for decomposing the hardware to fit the abstract machine.

DARTS provides an affinity class which gives the programmer the ability to partition the system into clusters based on the user's preference. Utilizing the hwloc library [8], the DARTS runtime determines how many cores and what levels of memory they share. When initializing the runtime the user may specify the number of clusters, and the CUs per cluster. Two modes of decomposing the system are offered.

- Spread mode - A single clusters is assigned to a single socket. Each socket has one TP scheduler and the requested number of CD schedulers. If there are not enough cores to satisfy the request (i.e. the user requested too many core per cluster or too many clusters), the affinity class will emit a warning, and use the full machine with the resources it has.
- Compact mode - Clusters are allocated one after another respecting no boundary (i.e. sockets or cache levels). This mode is useful for allocating more exotic divisions of a machine.

Using a configurable abstract machine provides the programmer flexibility while implementing applications, knowing he or she may configure the runtime to better exploit the inherent available parallelism. The two levels of parallelism provided by DARTS each present different means of implementing an application, and each have different runtime overheads which we explore in section 4.5. For some applications, switching between TP and codelet parallelism is simple, while for others it is very difficult. By reconfiguring the abstract machine, DARTS can better utilize the underlying hardware. This slack allows application programmers the ability to work with DARTS rather than against it, increasing programmability and performance.

4.2.1.4 Scheduling Policies

DARTS is designed to allow the exploration of several scheduling strategies. Both TP and CD schedulers may be specialized from their base scheduling class to provide numerous configurations. The primary components of a scheduling policy for the TP and CD scheduler are the following:

- Implementation of the pools (i.e. TPP, RP, and EP)
- How TPs and CDs are pushed and pulled from the pools
- Who pulls and pushes CDs from the RP

DARTS implements three main policies static, dynamic, and work stealing. All policies leverage Intel's TBB queues [41] to implement the pools.

4.2.1.4.1 Static

The static policy provided by DARTS permits the programmer to choose which core within a cluster will execute a codelet. This is done via the metadata field in the codelet class. The programmer uses this field to indicate the appropriate scheduler by choosing the id of the core in the cluster. Id zero is assigned to the TP scheduler, and is also a valid choice.

In the static policy the TP pool is implemented as a queue. TPs are pushed by running codelets into the TP scheduler's TP queue. When a TP scheduler is not

running any codelets, it attempts to acquire a TP from its queue. If it fails, the TP scheduler will try again from another TP scheduler's queue. Once the TP scheduler finds a TP, it will assign all enabled codelets to the schedulers indicated in metadata field of each codelet.

Each CD scheduler also has a queue which is used to hold enabled codelets. When the TP scheduler pushes the first enabled codelets to the CD scheduler, it places them here. The CD scheduler attempts to acquire an enabled codelet and fire it. If an executing codelet causes another codelet to become enabled, the newly enabled codelet is directly placed into the appropriate scheduler's codelet queue. In this way the RP is a distributed data structure.

This policy is intended for very regular applications. When applications have codelets which take varying amounts of time, it is difficult to fully utilize a system. If CD schedulers do not have enabled codelets waiting, they will spin locally waiting for work.

4.2.1.4.2 Dynamic

The dynamic scheduling policy stores all the enabled codelets in a single queue which is accessed by all the schedulers in a cluster. Each scheduler independently attempts to acquire work from the codelet queue including the TP scheduler.

The TP scheduler load balances TPs in the same manner as static scheduling using queues for the TPP. When the RP is empty, the TP scheduler will take a TP from the TPP. If there is no available TP, the TP scheduler will steal a TP from another random cluster. CD schedulers for the dynamic policy do not need their own codelet queues. Alternatively, the RP is a single queue located in the TP scheduler. Codelets are placed in the RP queue once they are enabled by either the TP scheduler or the CD scheduler. The codelets are dequeued and fired when a scheduler is free.

The dynamic policy is better suited for applications with non-uniform codelets. Additionally, CD schedulers have a known location to easily obtain work differing from random work stealing. This approach is sufficient when the number of schedulers in

the cluster is lower. The pressure and latency of queuing operations increases as the number of cores per cluster increases.

4.2.1.4.3 Work Stealing

The work stealing policy implements work stealing on both the TP and codelet level. TPs are load balanced by work stealing identical to the previous two policies. Each scheduler has a codelet queue which is used to store enabled codelets. The TP scheduler is responsible for initiating the execution of a TP. The TP scheduler places the initially enabled codelets of a TP into its codelet queue and proceeds to execute them. When CD schedulers do not have any codelets in their local queues, they randomly attempt to steal a codelet from another scheduler in the cluster including the TP scheduler's codelet queue. As codelets are executed, they enable other codelets. Once a codelet is enabled, it is placed into the codelet queue of the scheduler which enables it.

The work stealing policy is a more appropriate policy when the number of CD schedulers per cluster is higher and the application has many codelets with varying latencies. The pressure from the increased CD schedulers' queue operations is distributed by random stealing rather than focused on a single queue.

4.2.2 Use of Closures

The TP frame is an important mechanism which promotes the locality of shared data within a cluster. TPs are also a source of parallelism requiring load balancing to effectively use the underlying hardware. Moving data around the memory hierarchy is expensive in term of energy and time. To help reduce the cost of load balancing TPs between clusters, DARTS implements TP closures. A closure in DARTS is a data structure which contains the arguments passed to the TP and the type of TP (which in turn details the TP's function hence the name closure). A TP closure is smaller than a TP, since a it does not contain any codelets.

When a TP is invoked, a closure is created on the heap. That closure is initialized with the arguments of the TP, and placed in a TP scheduler's TPP. The closure can now be load balanced at a reduced cost. Once a TP scheduler is ready to run a TP, it is allocated, the closure's arguments are unmarshalled and placed into the TP's constructor, and the TP is initialized. Once the TP has been created, the closure is freed. This process is also done for loops, and is completely transparent to the programmer.

The alternative to this approach would be to immediately instantiate a TP creating the frame in memory near the allocating cluster. It is plausible that the TP may be stolen due to load imbalance. The cluster now executing the TP would have a longer latency accessing the TP frame including accessing codelets. Alternatively, DARTS ensures a TP is only allocated once it is ready to be executed, keeping it local.

4.2.3 Final Codelet

The DARTS runtime has a single entry and a single exit to and from the runtime. DARTS' exit is the runtime's final codelet. The final codelet is a special codelet with a single instance in the DARTS runtime. It is responsible for ending the execution of a codelet application and returning execution to the C++ code that began DARTS' execution.

Each scheduler is equipped with a local flag indicating the status of the runtime. During the construction of the runtime, the address of each scheduler's flag is given to the final codelet. The final codelet has a synchronization slot count of one. When the final codelet's slot is decremented, the codelet signals each scheduler that the execution is complete. In order to reduce spinning on these flags, each scheduler will perform all of its duties before checking the flag (e.g. a CD scheduler will execute all of the codelets in its local pool before checking its flag). The result of the application is not guaranteed when codelets race to signal the final codelet. After executing the final codelet, all the schedulers spin waiting to run another DARTS application except one which returns to the code which launched the DARTS application. The runtime will persist until the runtime destructor is call.

4.3 API

While the majority of an application written for DARTS is comprised of inherited objects (e.g. codelets, TPs, and loops), a handful of API calls are still necessary to interface with the DARTS runtime. These functions include instantiating and configuring the runtime, starting execution, invoking new TPs, and satisfying events. The following section describes and provides examples for how to perform these operations.

4.3.1 Runtime

As previously mentioned the runtime is implemented as a single object. The runtime object creates threads and pins them to cores. By default, the runtime decomposes the system based on the number of sockets in the system. Alternatively, one can provide the runtime with the number of clusters and CD schedulers per cluster. The following is an example of instantiating the runtime.

Listing 4.1: The runtime instance rt will partition the hardware based on the number of sockets in the system.

```
#include <iostream>
#include ``darts.h''
using namespace darts;
...
Runtime rt;
...
```

Listing 4.2: The runtime instance rt will have 4 clusters with 3 CD schedulers each totaling 16 schedulers.

```
#include <iostream>
#include ``darts.h''
using namespace darts;
...
```

```
  Runtime rt(4,3);
...
```

4.3.2 Affinity

In order to decompose the hardware as specified in section 4.2.1.3, DARTS provides an affinity object. This object permits several options including the number of clusters, number of CD schedulers per cluster, the affinity mode (spread or compact as described in section 4.2.1.3), and the policies for both TP and CD schedulers. These parameters are passed into the affinity class, and a mask is generated via generateMask method. This method returns a boolean value indicating whether the input parameters provide a valid decomposition of the system. An example of an invalid decomposition for a 48 core system would be to request 8 clusters with 8 cores per cluster. After successfully generating the mask, the affinity class is passed into the runtime constructor to create the desired runtime configuration. The following is an example.

```
#include <iostream>
#include ``darts.h''
using namespace darts;
...
  //Generates the affinity class with appropriate parameters
  ThreadAffinity affinity(3, 2, COMPACT, TPDYNAMIC, MCDYNAMIC);

  //Ensure the parameters are correct
  if (affinity.generateMask())
  {
    //Pass the affinity class into the runtime constructor
    Runtime rt(&affinity);
  }
...
```

4.3.3 Launch/Invoke

An application written in DARTS is comprised of TPs and Codelets. The invoked TPs are provided to the runtime through two different methods which rely on C++'s templating functionality. The first method is used to run the initial TP. The launch function takes a template parameter of the type of TP to instantiate. Additionally, the TP's parameters are passed as regular parameters to the launch function identically to the required parameters of the TP. The launch function returns a TP closure which is passed into the runtime's run method. The following is an example of using the launch method.

```
#include <iostream>
#include ``darts.h''
using namespace darts;

class ExampleTP : public ThreadedProcedure
{
  ...
  //TP constructor and its required parameters
  ExampleTP(int Number, Codelet * ToSignal);
  ...
};
...
  //Using runtime run method with launch to start a fib TP
  Runtime().run(launch<ExampleTP>(10, &Runtime::finalSignal));
...
```

The remaining TPs are instantiated during the execution of codelets via the invoke method. The invoke method is identical to the launch method except it requires the first function parameter to be the address of the invoking TP. This distinction is necessary for DARTS to automatically delete TPs. The following is an example of a codelet using the invoke method.

```
#include <iostream>
#include ``darts.h''
```

```
using namespace darts;

class ExampleTP : public ThreadedProcedure
{
  ...
  //TP constructor and its required parameters
  ExampleTP(int X, int Y);
  ...
};
...
void CodeletExample::fire(void)
{
  invoke<ExampleTP>(myTP_, 5, 6);
  ...
}
...
```

4.3.4 Signaling Codelets

Each codelet has a synchronization slot used to track of the required events to enable a codelet. To satisfy an event, one must use the decDep (decrement dependency) method. The decDep method requires no parameters, and is responsible for decrementing the codelet's synchronization slot by one. If the synchronization slot reaches zero the codelet is placed into the RP by the codelet responsible for decrementing it. The following is an example of using the decDep method.

```
#include <iostream>
#include ``darts.h''
using namespace darts;
...
class ExampleTP : public ThreadedProcedure
{
  CodeletOne cd1;
  CodeletTwo cd2;
  ...
```

```
};

void cd1::fire(void)
{
  exampleTP * myExampleTP = static_cast<ExampleTP*>(myTP_);
  std::cout << "Hello!" << std::endl;
  myExampleTP->cd2.decDep();
}
```

4.4 Example

Until now, we have shown the pieces of the DARTS runtime, but have not demonstrated a complete program. The following section will explore a simple and common program to provide a more complete view. Calculating Fibonacci numbers is a trivial recursive program. A naive implementation is provided below.

```
#include <iostream>

int fib(int Number)
{
  if(Number<2)
    return Number;
  else
  {
    int x = fib(Number-1);
    int y = fib(Number-2);
    return x + y;
  }
}

int main(void)
{
  std::cout << fib(10) << std::endl;
  return 0;
}
```

The fib function takes a single parameter, an integer. If the number is greater than two, fib is recursively called for the integer minus one and minus two. The results of these calls are summed. If the inputted number is less than two, the number is returned.

The first step in writing this program for DARTS is to decompose the fib function into codelets. The fib function itself will serve as the TP. Two parts of fib are immediately visible. The first part checks the input and conditionally performs the recursive call. The second performs the sum and returns the result. These two parts will be the basis for two codelets. This forms a split-phase transaction as described in 3.2.3.1. The following is the Fibonacci example written for DARTS.

```
1  #include <iostream>
2  #include ``darts.h''
3  using namespace darts;
4
5  //This codelet is responsible for the conditional recursive calls
6  class FibCheck : public Codelet
7  {
8  public:
9    FibCheck(uint32_t Dependecy, uint32_t Reset, ThreadedProcedure *
       MyTP, uint32_t Metadata):
10   Codelet(Dependecy,Reset,MyTP,Metadata) { }
11
12   virtual void fire(void);
13 };
14
15 //This codelet is sums the two recursive calls and writes the result
16 class FibAdd : public Codelet
17 {
18 public:
19   FibAdd(uint32_t Dependecy, uint32_t Reset, ThreadedProcedure * MyTP
       , uint32_t Metadata):
20   Codelet(Dependecy,Reset,MyTP,Metadata) { }
21
```

```
22    virtual void fire(void);
23  };
24
25  //This is the fib TP
26  class Fib : public ThreadedProcedure
27  {
28  public:
29    int number; //This is the inputed number
30    int x; //This is the result of the first recursive call
31    int y; //This is the result of the second recursive call
32    int * result; //This is where to write the address
33    FibCheck check; //The instantiated fibCheck codelet
34    FibAdd adder; //The instantiated fibAdd codelet
35    Codelet * toSignal; //This is who to signal when complete
36    Fib(int Number, int * Result, Codelet * ToSignal):
37    ThreadedProcedure(),
38    number(Number),
39    x(0),
40    y(0),
41    result(Result),
42    check(0, 0, this, 0), //This codelet has no required events
43    adder(2, 2, this, 0), //This codelet has two required events
44    toSignal(ToSignal)
45    {
46        add(&check);
47    }
48  };
49
50  void FibCheck::fire(void)
51  {
52    fib * myFib = static_cast<Fib*>(myTP_);
53    if(myFib->number<2)
54    {
55      (*myFib->result) = myFib->number;
56      myFib->toSignal->decDep();
```

```
57    }
58    else
59    {
60      invoke<Fib>(myFib, myFib->num-1, &myFib->x, &myFib->adder);
61      invoke<Fib>(myFib, myFib->num-2, &myFib->y, &myFib->adder);
62    }
63  }
64
65  void FibAdd::fire(void)
66  {
67    fib * myFib = static_cast<Fib*>(myTP_);
68    (*myFib->result) = myFib->x + myFib->y;
69    myFib->toSignal->decDep();
70  }
71
72  int main(void)
73  {
74    int result;
75    ThreadAffinity affinity(3, 2, COMPACT, TPDYNAMIC, MCDYNAMIC);
76    if (affinity.generateMask())
77    {
78      Runtime(&affin).run(launch<fib>(10, &result, &Runtime::
      finalSignal));
79      getTime(&end);
80      std::cout << "fib 10 " << result << std::endl;
81    }
82    else
83      std::cout << "Invalid configuration" << std::endl;
84    return 0;
85  }
```

4.4.1 Fibonacci TP

The two codelets of the Fibonacci function are declared on lines 6 and 16. The Fib TP instantiates these codelets on lines 33 and 34 as check and adder. The check

codelet requires no events before it can fire. This codelet is given to the runtime by "adding" it as done on line 46. The Fib TP frame has five members which are shared by the codelets including the input, two intermediates values, and an address to write the result. The final member is a codelet pointer used to indicate which codelet to signal at the completion of the TP.

4.4.2 Fibonacci Codelets

The FibCheck codelet is responsible for checking if the inputed integer is less than two. To check the input, the codelet uses its `myTP_` member. To properly access the members of the TP, the `myTP_` member must be cast into the proper TP type done on line 52. Line 53 checks if the input is less than two. If it is, the result is written to the address stored in the result member of the TP on line 55. Next, the codelet passed to the TP is signaled completing the codelet and the TP. If the input is greater than or equal to two, the codelet invokes two new TPs on lines 60 and 61. The invoke has four inputs, the first is its parent TP. The remaining arguments to the TP are the inputs to a fib TP. The result addresses are the x and y field of the parent TP. When the children TP finish, they will signal the FibAdd codelet. The invoked TPs are executed differently than traditional recursion. The invoke methods do not block, but rather they push TP closures into the TPP and continue executing fibCheck.

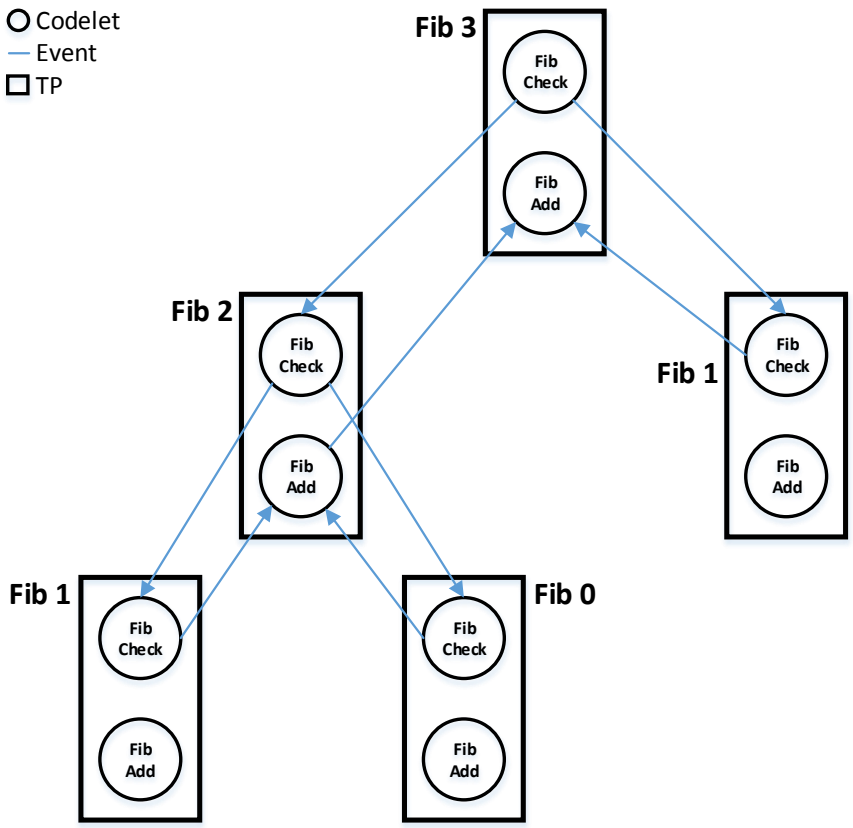


Figure 4.4: This is shows the Fibonacci function in codelets and TPs. This example show the calculation the 3rd Fibonacci number. The FibCheck codelet is responsible for checking if the Fibonacci Number is less than 2, and invoking Fibonacci Number-1 and Number-2. Otherwise, FibCheck writes the result and signals its completion. FibAdd is signaled only if it is a internal TP in the Fib TP invocation tree. It is signaled by its children TPs once they have written their results into the their TP frame. Once enabled, FibAdd sums the two results, stores them in its parent's TP frame, and signals its completion.

The FibAdd codelet requires two events before it can execute. These events

represent the availability of x and y. These two members are filled by the the recursive TP invocations. Once x and y have been written, FibAdd sums them together, and writes it to the result address. On line 69, FibAdd signals its completion.

4.4.3 Runtime

The runtime is configured using an affinity class on line 75. The runtime has 2 clusters each with 3 codelet schedulers. The runtime will configure 8 schedulers using the compact distribution running the dynamic scheduling policies. Line 76 generates the mask and ensures it is valid. The runtime launches the initial Fib TP calculating the 10th Fibonacci number. The result is stored in the result integer declared on line 74. The runtime will execute until the initial TP launched signals the final codelet. The application concludes by displays the results.

4.5 Micro Benchmarks

The remainder of this chapter is dedicated to understanding the overheads of the DARTS runtime. We examine the overheads in initializing the system and using both codelets and TPs. Furthermore, we show the overhead of several common patterns. To do so, we use two different x86 shared memory systems.

4.5.1 Mills

The Mills system has 48 AMD cores operating at 2.4 GHz. These cores are divided between four AMD Opteron 6234 (Interlagos) processors. The system is equipped with 4 x 32 GB of DDR3-1333 ECC memory. Each core has its own 16 KB L1 data cache. Two cores share a 2 MB L2 unified cache, and six cores share a unified 6 MB L3 cache. A floating-point unit is shared between two cores, and is capable of executing up to four double precision operations at once using AVX instructions. The Mills system runs Scientific Linux 6. DARTS and all of the kernels presented are compiled with GCC version 4.6.2 using an optimization level -O3. All experiments presented decompose the Mills system into 4 clusters each with 1 TP scheduler and 11 CD schedulers using the spread configuration.

4.5.2 Monica

Our second testbed is the 24 core Monica system. This system is comprised of 4 Intel Xeon E5-4610 processors each containing 6 cores. Each core is hyper-threaded providing in total 48 threads. Every core has its own 32 KB L1 data cache and a unified 256 KB L2 cache. A unified 15 MB L3 cache is shared by six cores. This system is also equipped with 4 x 32 GB of DDR3-1333 ECC memory. For this system we use GCC version 4.7.3 with an optimization level of -O3 to compile DARTS and the kernels. Furthermore, Monica runs Ubuntu 13.04. All experiments presented decompose the Monica system into 4 clusters each with 1 TP scheduler and 11 CD schedulers using the spread configuration.

4.5.3 Runtime

Initializing the runtime consists of launching threads, initializing the schedulers, and organizing the schedulers into clusters. Figure 4.5 shows the cost of initializing the runtime using different configurations.

While the primary component in runtime initialization is the cost of creating the underlying pthreads, overhead is also added by connecting all the schedulers together. As TP schedulers are added, they must be visible to all other TP schedulers. When CD schedulers are added, they are only required to be connected to TP scheduler in their cluster. The overhead of initializing the runtime is only incurred once for a given application, as a runtime can be reused.

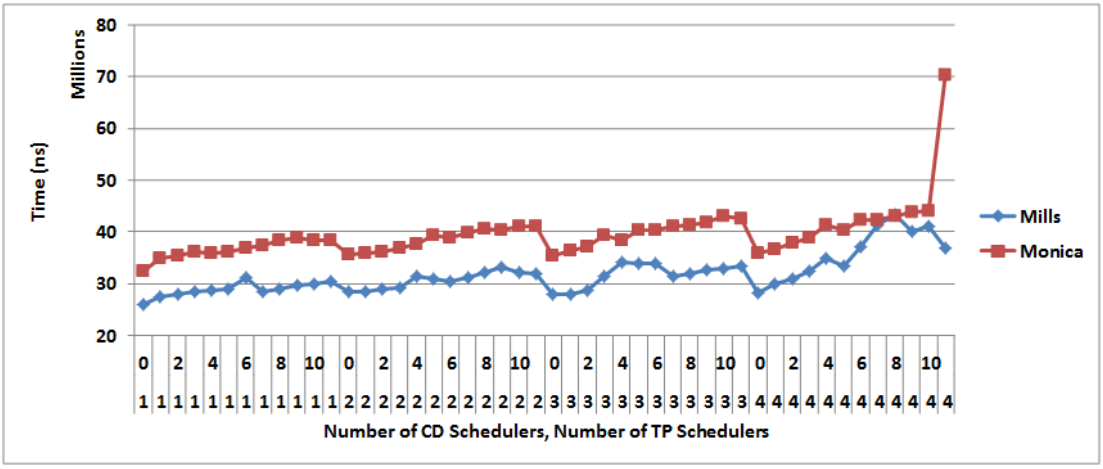


Figure 4.5: Cost of initializing the DARTS runtime in nanoseconds scaling both the number of TP Schedulers and CD Schedulers. The X axis show two numbers per data point. The bottom number is the amount of clusters (1-4). The top is the number of CD schedulers *per* cluster (0-11).

4.5.4 TP

Next we explore the overhead involved in using a TP. Since both the invoke and launch functions are nearly identical, we present only the cost of launching a single TP with a single member, the address of the final codelet. To observe this overhead without interference from a particular scheduling policy, we utilize only a single TP scheduler. Table 4.1 presents the results from both the Mills and Monica systems.

Mills	Monica
293 ns	171 ns
703.2 cycles	410.4 cycles

Table 4.1: Overhead of launching a single TP with only one member, the address of the final codelet.

Using both the collected time in nanoseconds and the maximum frequency of both systems, we are capable of giving an upper bound to the number of cycles it takes to use a TP which is presented below the time. These overheads include several important operations

- Construction and destruction of a TP closure
- Queuing and dequeuing the TP closure into the TP scheduler's TP Ready pool
- The construction of the TP and the unmarshalling of its arguments from the TP closure into the TP frame

It is important to note that as more arguments are added to the TP's invocation, the longer it will take to marshal and unmarshal these arguments to and from the TP closure and TP frame.

4.5.5 Codelet

The cost of creating codelets is typically included in the cost of constructing the TP since codelets are members of the TP class. Once created, the TP constructor places ready codelets into the codelet ready pool. To observe the overhead of a codelet we present two different measurements. The first measures the total time including the construction of the TP and the codelet, pushing the codelet to a scheduler, and running the codelet. In order to measure the overhead alone, the codelet only signals its completion. Only a single scheduler is used to avoid any interference from the scheduling policies. The following table 4.2 presents the total time running only a single codelet.

Mills	Monica
456 ns	249 ns
1094.4 cycles	597.6 cycles

Table 4.2: CD overhead measuring total time including creation, scheduling, and execution of an empty codelet.

The second measurement only reports the costs of pushing and pulling the codelet to the ready pool and running the empty codelet. Table 4.3 presents this time.

Mills	Monica
254 ns	110 ns
609.6 cycles	264 cycles

Table 4.3: CD overhead measuring only scheduling and execution of an empty codelet.

The overhead incurred from push and pulling the codelet to and from the RP is directly dependent on the data structure used to implement the RP.

4.5.6 Codelet Fanout

One very common codelet pattern that arises in many applications is a fanning out of codelets from a source. These codelets typically signal a single codelet upon completion (sink). This pattern can also be interpreted as a for all codelet loop as described in section 4.1.3.2. In the following section we explore this pattern and its overheads.

The fanout pattern we use is comprised of a single TP, with a start and end codelet surrounding several parallel codelets. The parallel codelets are dynamically allocated during the TP's construction. These codelets perform no work, only signaling their completion to the end codelet. Since the codelets are contained only in a single TP, they can only be executed on a single cluster.

We evaluate the time taken by executing the fanout pattern while varying the number of parallel codelets, machine configuration, and scheduling policy presenting (without the overhead of creating the runtime). The policies used are presented in section 4.2.1.4. The static policy assigns each parallel codelet to a scheduler in a round robin fashion while the start and end codelet are assigned to the TP scheduler.

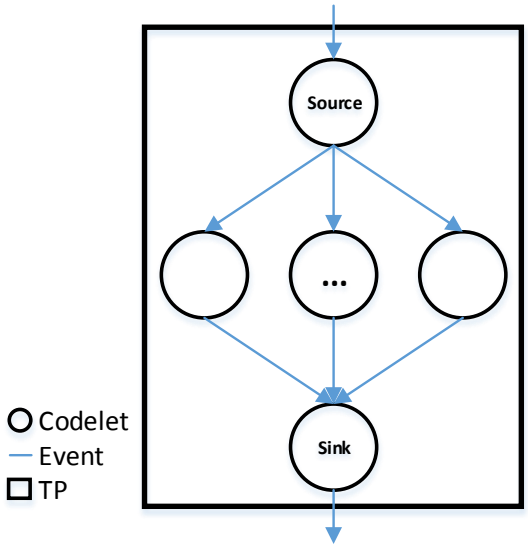
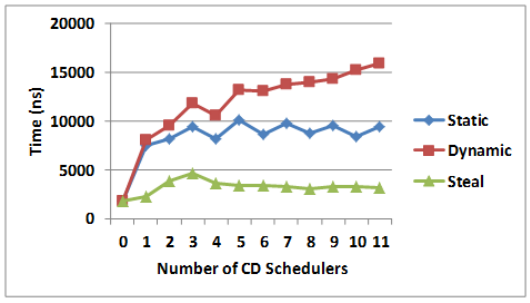


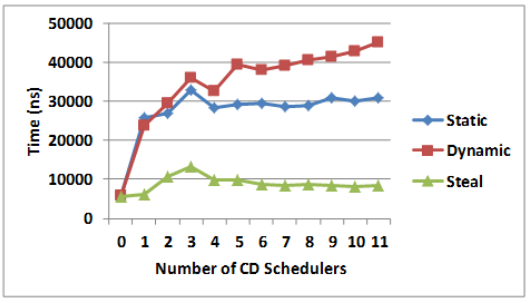
Figure 4.6: The codelet fanout pattern begins execution with a signal codelet, the source. This codelet signals from 0 to N codelets which may execute in parallel. The execution completes with a sink codelet which fires after the N parallel codelets signal.

4.5.6.1 Mills

Figures 4.7(a) and 4.7(b) present the results of executing the fanout pattern on the Mills system with fanouts of 8 and 32 respectively. We present these results executing on a single cluster (using only a single TP scheduler) and scaling the number of CD schedulers (i.e. CUs). For an application with codelets containing actual work, the total time should decrease as more CD schedulers are added. However, by using empty codelets, we only increase the overheads associated with using more schedulers. This is most evident in the Dynamic policy. As more CD schedulers are added, more pressure is placed on the codelet ready pool. Conversely, the static policy remains more stable since there is only contention between the TP scheduler pushing and a single



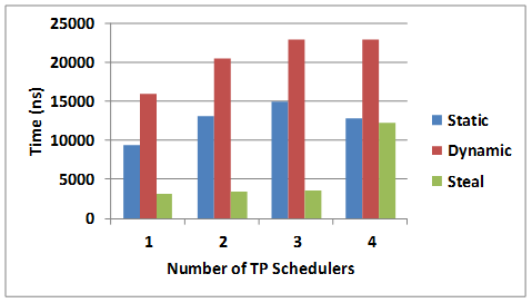
(a) The fanout pattern of 8 codelets



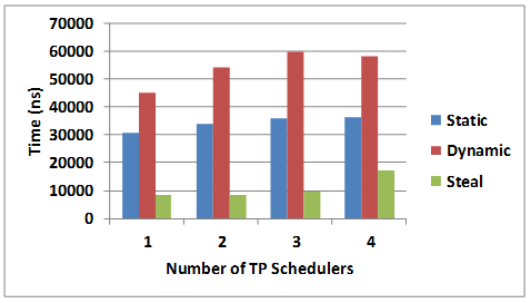
(b) The fanout pattern of 32 codelets

Figure 4.7: These graphs present the codelet fanout pattern running on the Mills system scaling the use a single cluster.

CD scheduler pulling codelets. The steal policy suffers a significantly reduced penalty for adding CD schedulers since their access are distributed across the cluster.



(a) The fanout pattern of 8 codelets

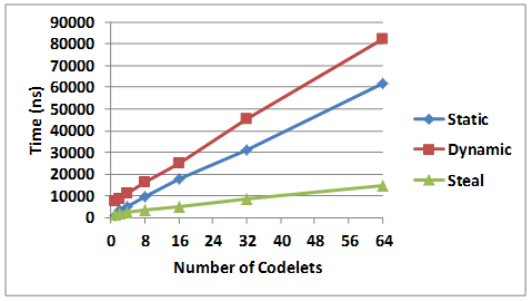


(b) The fanout pattern of 32 codelets

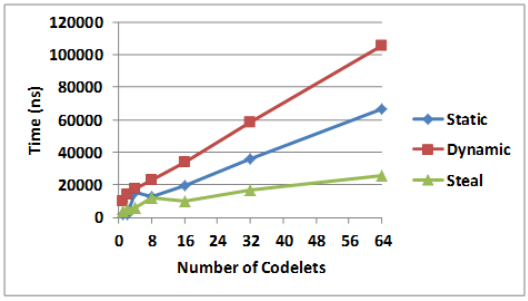
Figure 4.8: These graphs present the codelet fanout pattern running on the Mills system using 11 CD schedulers per cluster.

Figures 4.8(a) and 4.8(b) illustrate adding multiple fully utilized clusters (each with 11 CD schedulers). While the benchmark only uses one TP capable, an interesting effect is observed. As more TP schedulers are added, the execution time is increased for all policies. This is due to the identical TP load balancing (stealing) performed by all of the policies. As TP schedulers are added pressure is added on the TP ready pool.

Moreover, the single TP may be stolen once between the time the TP is released into the runtime and the when the first TP scheduler checks its ready pool.



(a) The fanout pattern using 1 full cluster



(b) The fanout pattern using the full system

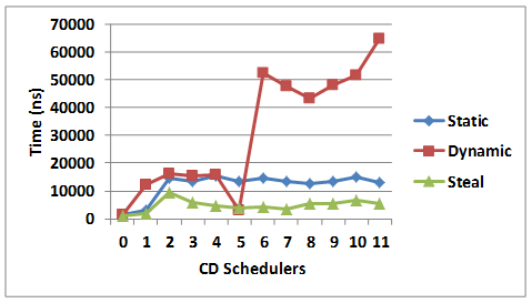
Figure 4.9: These graphs present the codelet fanout pattern running on the Mills system while scaling the number of codelets.

Lastly we present the execution time taken while scaling the number of parallel codelets. Figures 4.9(a) and 4.9(b) present this scaling using two configurations, a single cluster and the entire machine respectively. This view of the system depicts the linearly increasing overhead of creating more codelets and adding them to the runtime.

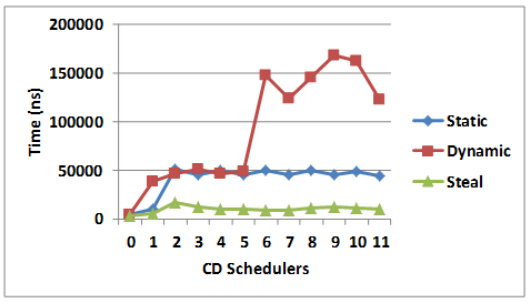
4.5.6.2 Monica

Figures 4.10(a) and 4.10(b) present the fanout of 8 and 32 codelets running on the Monica system using a single cluster. Again, we scale the number of CD schedulers used in the cluster. The steal policy incurs the least overhead in both cases. In addition, the static policy has a more constant overhead which increases as more codelets are scheduled in parallel. The dynamic policy exhibits low overhead when using half a cluster or less. Performance suffers as more threads are used because Intel's hyper-threading splits a core's resources (cache and execution pipeline) between two threads.

Figures 4.11(a) and 4.11(b) show the effects of adding clusters. The steal policy slowly increases as more clusters are added similarly to the Mills system. The dynamic and static policy do not however as we increase the number of clusters.



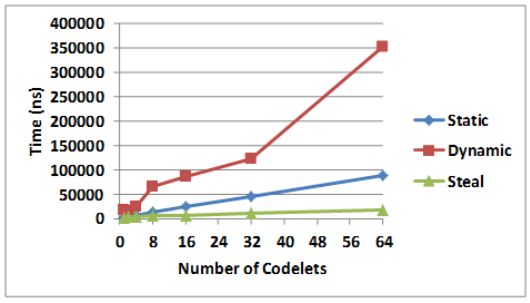
(a) The fanout pattern of 8 codelets



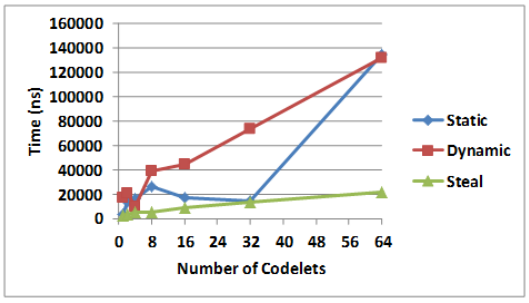
(b) The fanout pattern of 32 codelets

Figure 4.10: These graphs present the codelet fanout pattern running on the Monica system scaling the use of 1 cluster.

Figures 4.12(a) and 4.12(b) show how the policies perform as more parallel codelets are added. The steal policy is consistent and scales the best of the three. Moreover, using a single cluster scales similarly to the Mills system.

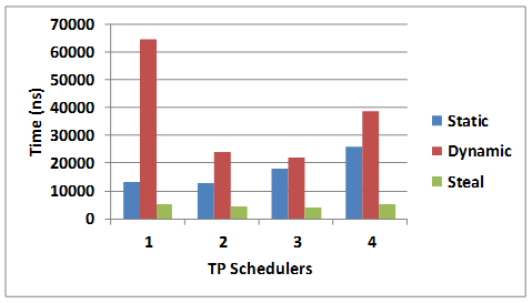


(a) The fanout pattern using 1 full cluster

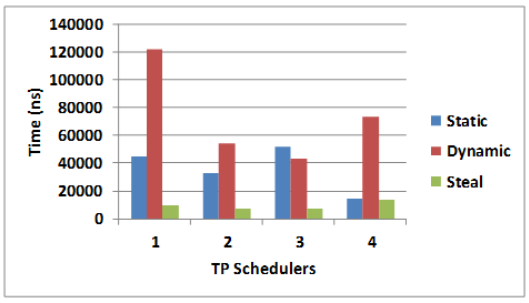


(b) The fanout pattern using the full system while scaling the number of codelets

Figure 4.12: These graphs present the codelet fanout pattern running on the Monica system while scaling the number of codelets.



(a) The fanout pattern of 8 codelets



(b) The fanout pattern of 32 codelets

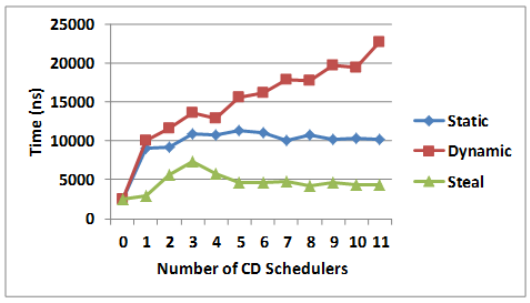
Figure 4.11: These graphs present the codelet fanout pattern running on the Monica system using 11 CD schedulers per cluster.

4.5.6.3 Codelet For All Loop

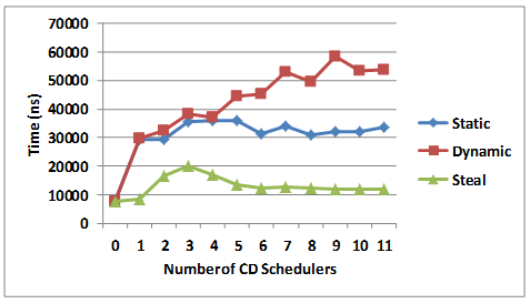
The fanout pattern is almost identical to the codelet loop presented in section 4.1.3.2 providing us an opportunity to compare the two. The codelet for all loop only differ slightly from the fanout pattern in that the starting codelet is also the finishing codelet. For the static scheduling policy, each iteration is assigned in a round robin fashion to the schedulers within a cluster.

Figure 4.13(a) through 4.13(f) present the same set of experiments run on the Mills cluster as in section 4.5.6.1. The results show the same trends as before with a slight increase in overhead. The additional overhead incurred can be attributed to the addition of the loop construct which requires construction and argument demarshalling.

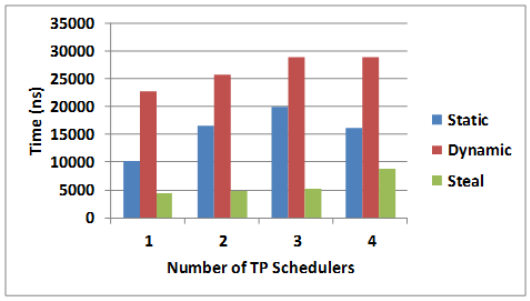
Figure 4.14(a) through 4.14(f) presents the experiments run on the Monica cluster. The steal and static policies both have a slightly greater overhead. The dynamic policy however does not exhibit the same behavior as the fanout pattern. Instead the policy scales linearly as the number of CD schedulers are added. This same difference can be seen through the remainder of the experiments.



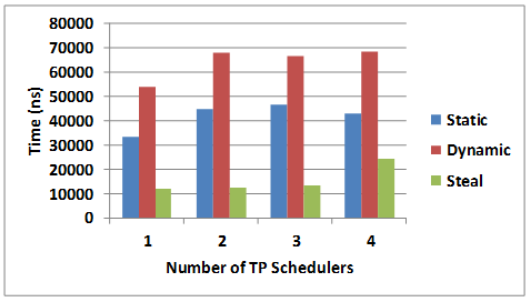
(a) Codelet for loop of 8 codelets scaling the use of 1 cluster



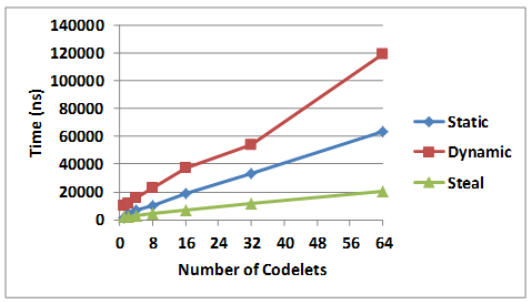
(b) Codelet for loop of 32 codelets scaling the use of 1 cluster



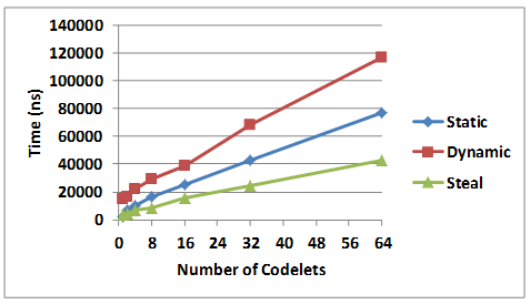
(c) Codelet for loop of 8 codelets using 11 CD schedulers per cluster



(d) Codelet for loop of 32 codelets using 11 CD schedulers per cluster

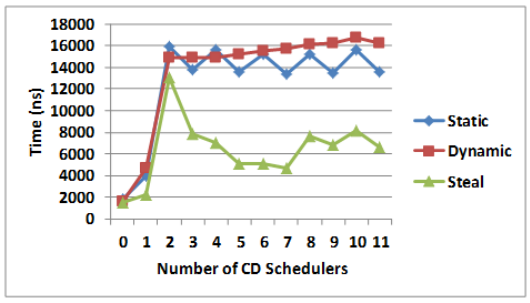


(e) Codelet for loop using 1 full cluster while scaling the number of codelets

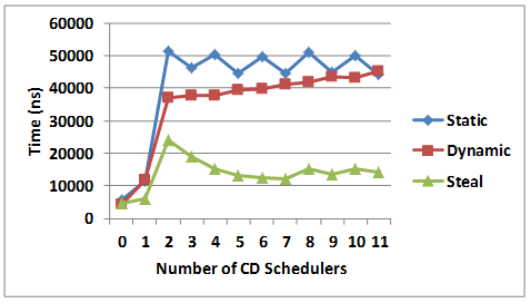


(f) Codelet for loop using the full system while scaling the number of codelets

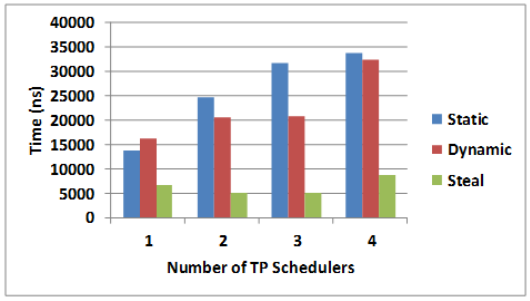
Figure 4.13: These graphs present the codelet for loop running on the Mills system.



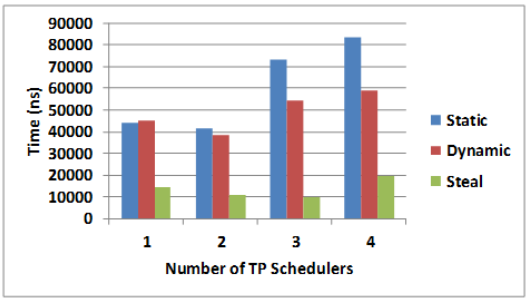
(a) Codelet for loop of 8 codelets scaling the use of 1 cluster



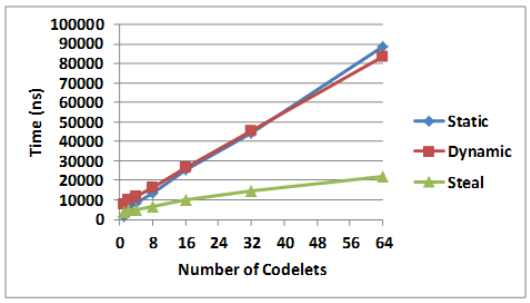
(b) Codelet for loop of 32 codelets scaling the use of 1 cluster



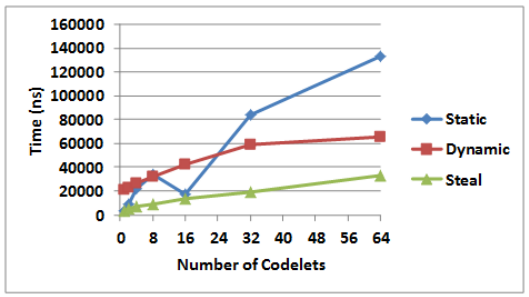
(c) Codelet for loop of 8 codelets using 11 CD schedulers per cluster



(d) Codelet for loop of 32 codelets using 11 CD schedulers per cluster



(e) Codelet for loop using 1 full cluster while scaling the number of codelets



(f) Codelet for loop using the full system while scaling the number of codelets

Figure 4.14: These graphs present the codelet for loop running on the Monica system.

4.5.7 TP Fanout

In addition to exploring the fanout of codelets, we also investigate the fanout of TPs. Codelet applications are more likely to fanout using TPs when there are either larger amounts of work to do (multiple codelets) or work needs to be distributed beyond a single cluster. Exploring the TP fanout pattern sheds light on the overheads of load balancing the entire system. The TP fanout pattern begins with a single TP with two codelets acting as the source and the sink. The source codelet invokes several parallel TPs each containing one codelet. When the parallel TP's codelet is fired, it signals the initial TP's sink. When all of the TPs have finished, the sink signals the final codelet and the benchmark is completed. The static scheduling policy assigns the single codelet to the TP scheduler of each cluster.

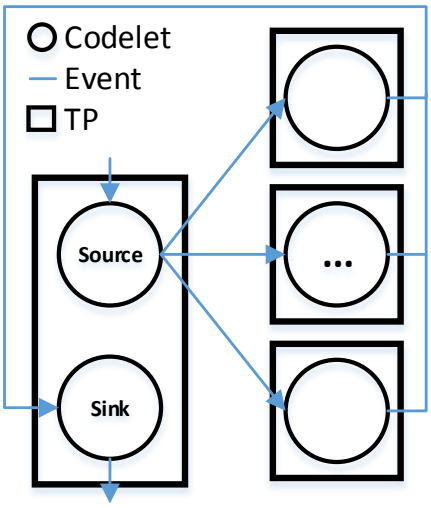


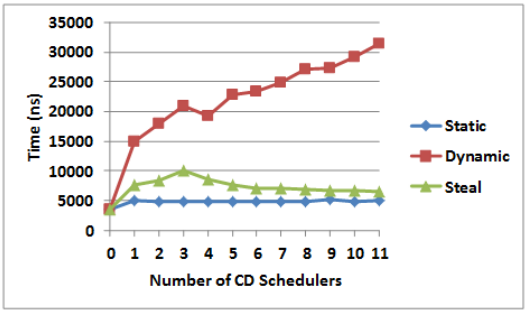
Figure 4.15: The TP fanout pattern begins with the a signal codelet which invokes 0 through N TPs each with a signal codelet. These children TPs execute in parallel and signal their completion to a sink in the parent TP.

4.5.7.1 Mills

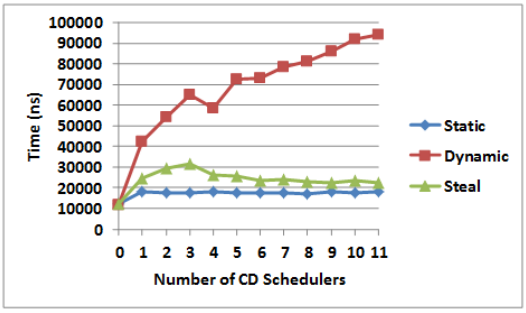
We present the results of executing the TP fanout pattern in figures 4.16(a) through 4.16(f). Figures 4.16(a) and 4.16(b) show the overhead involved in scaling the number of CD schedulers using a single cluster demonstrating a similar pattern as the CD fanout pattern. Figures 4.16(c) and 4.16(d) scale the number of TP schedulers without using any CD schedulers. Since no useful work is being executed, using more than one cluster only adds overhead. However we see as more clusters are added this overhead is decreased since TPs may run in parallel. Lastly figure 4.16(e) and 4.16(f) show linear scaling as more TPs are executed.

4.5.7.2 Monica

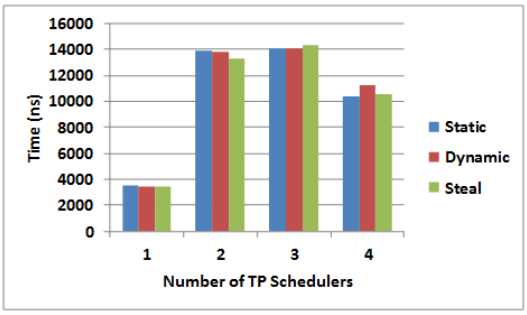
Executing the TP fanout pattern on the Monica system provides similar results. Figures 4.17(a) and 4.17(b) show the same differences between scheduling policies as the results from the Mills system. The dynamic policy running on Monica does differ from Mills in that it does not suffer increasingly larger overheads as CD schedulers are added. Rather it maintains a more constant overhead close to the maximum overhead. Figures 4.17(c) and 4.17(d) again demonstrated the diminishing overhead in adding clusters. Lastly, via figures 4.17(e) and 4.17(f) we see that all policies scale roughly linearly as more TPs are added.



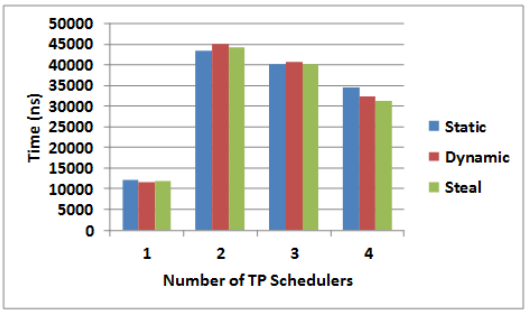
(a) The fanout of 8 TPs scaling the use of 1 cluster



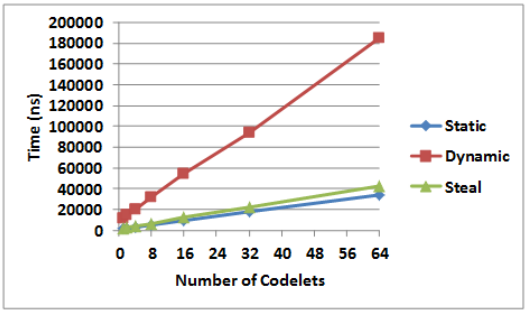
(b) The fanout of 32 TPs scaling the use of 1 cluster



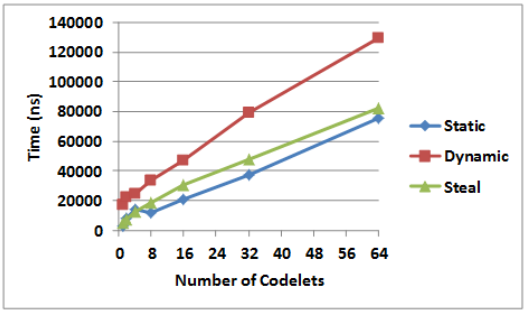
(c) The fanout of 8 TPs using only 1 TP scheduler per cluster (no CD schedulers)



(d) The fanout of 32 TPs using only 1 TP scheduler per cluster (no CD schedulers)

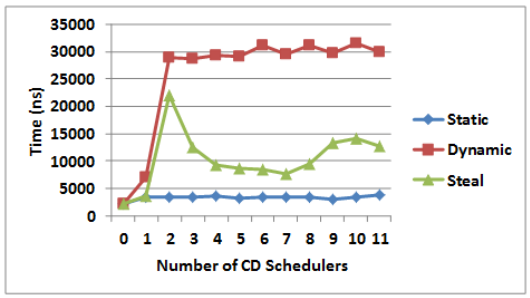


(e) Fanout using 1 full cluster while scaling the number of TPs

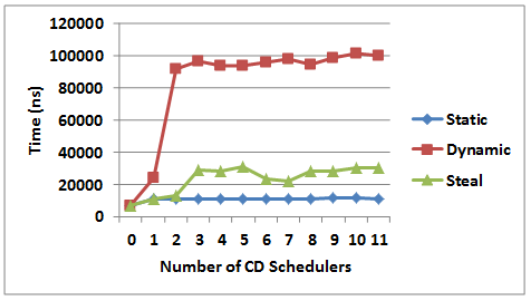


(f) Fanout using the full system while scaling the number of TPs

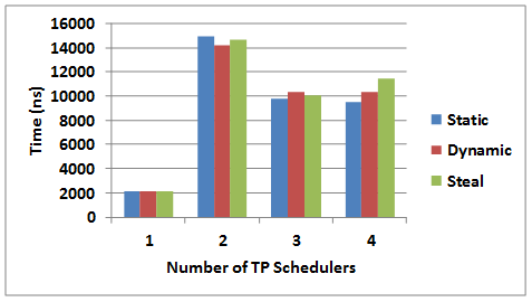
Figure 4.16: These graphs present the TP fanout pattern running on the Mills system.



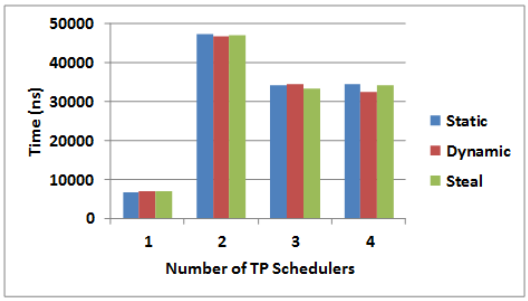
(a) The fanout of 8 TPs scaling the use of 1 cluster



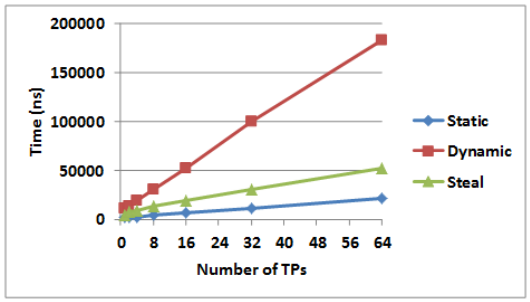
(b) The fanout of 32 TPs scaling the use of 1 cluster



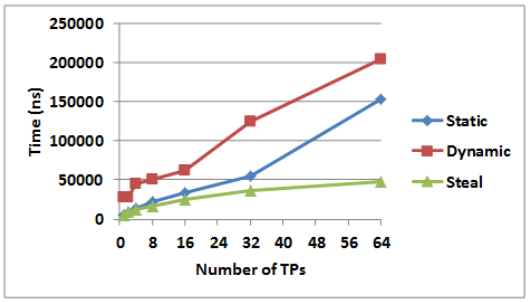
(c) The fanout of 8 TPs using only 1 TP scheduler per cluster (no CD schedulers)



(d) The fanout of 32 TPs using only 1 TP scheduler per cluster (no CD schedulers)



(e) Fanout using 1 full cluster while scaling the number of TPs

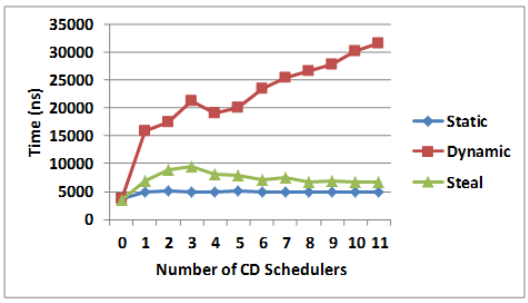


(f) Fanout using the full system while scaling the number of TPs

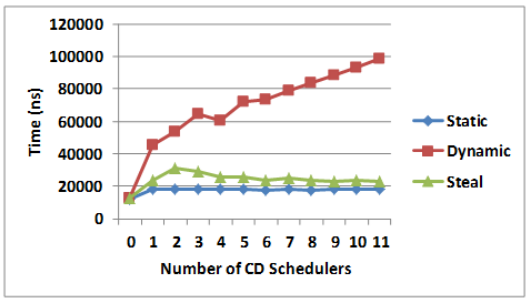
Figure 4.17: These graphs present the TP fanout pattern running on the Monica system.

4.5.7.3 TP For All Loop

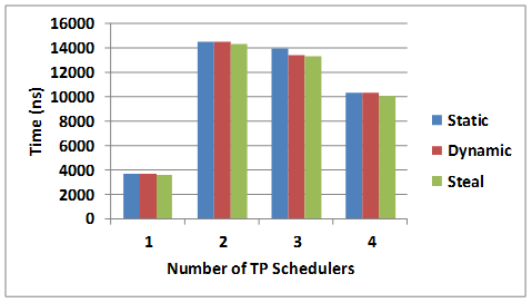
Once again we can draw a comparison of the TP fanout pattern and the TP for all loop. The TP for all loop differs from the codelet for all loop as all of its iterations may be executed on multiple clusters just as the TP pattern. Figures 4.18(a) through 4.18(f) present the same experiments as before using the TP for all loop rather than the TP fanout pattern running on the Mills system. The same patterns emerge with an almost negligible difference. Figures 4.19(a) through 4.19(f) present the results for the Monica system using the TP for all loop. The difference in overhead between the TP for all loop and the TP fanout pattern on Monica is slightly more pronounced, however the same trends still appear.



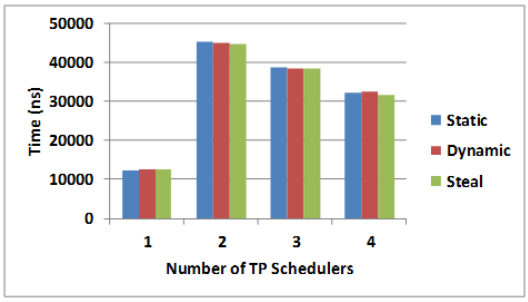
(a) TP for loop of 8 TPs scaling the use of 1 cluster



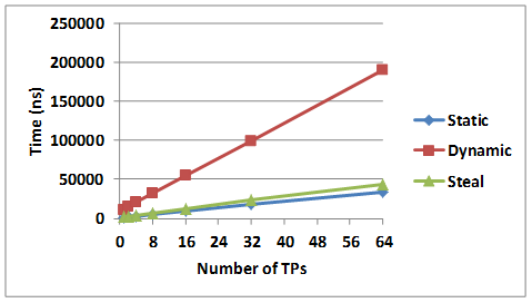
(b) TP for loop of 32 TPs scaling the use of 1 cluster



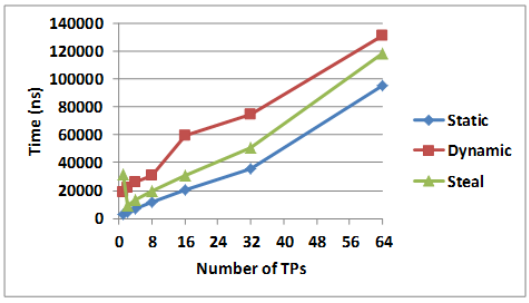
(c) TP for loop of 8 TPs using only 1 TP scheduler per cluster (no CD schedulers)



(d) TP for loop of 32 TPs using only 1 TP scheduler per cluster (no CD schedulers)

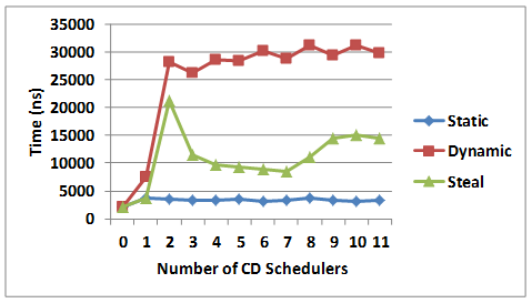


(e) TP for loop using 1 full cluster while scaling the number of TPs

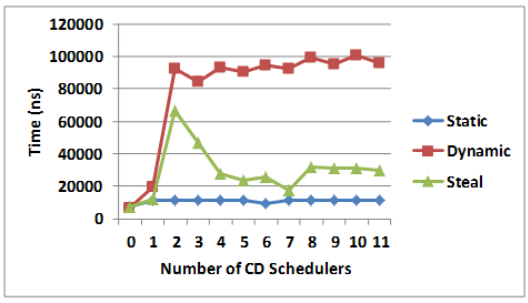


(f) TP for loop using the full system while scaling the number of TPs

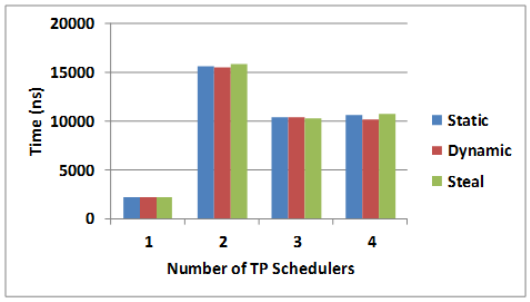
Figure 4.18: These graphs present the TP for loop running on the Mills system.



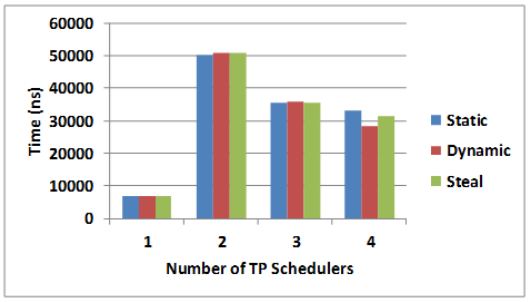
(a) TP for loop of 8 TPs scaling the use of 1 cluster



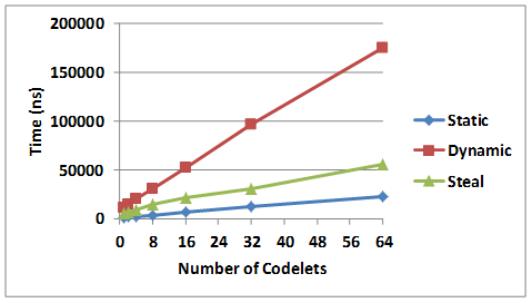
(b) TP for loop of 32 TPs scaling the use of 1 cluster



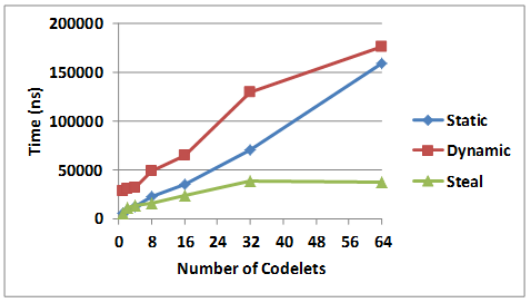
(c) TP for loop of 8 TPs using only 1 TP scheduler per cluster (no CD schedulers)



(d) TP for loop of 32 TPs using only 1 TP scheduler per cluster (no CD schedulers)



(e) TP for loop using 1 full cluster while scaling the number of TPs



(f) TP for loop using the full system while scaling the number of TPs

Figure 4.19: These graphs present the TP for loop running on the Monica system.

4.5.8 Codelet Chain

The codelet chain pattern is another very common pattern which arises in codelet applications. This pattern is comprised of sequentially executed codelets. Using this benchmark we can observe the cost of signaling codelets within a cluster. This benchmark consists of a single TP with several codelets which signals each other one after another. The static policy assigns all of the codelets to a single scheduler.

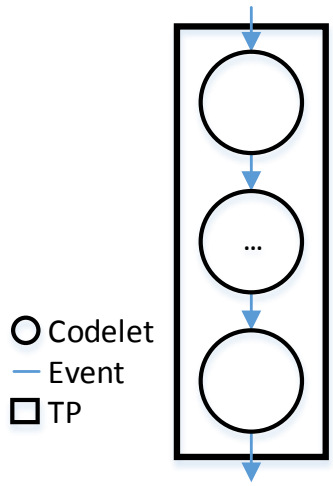
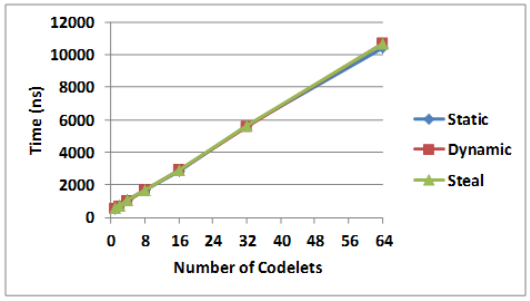


Figure 4.20: The CD chain pattern consists of a group of serially executed codelets.

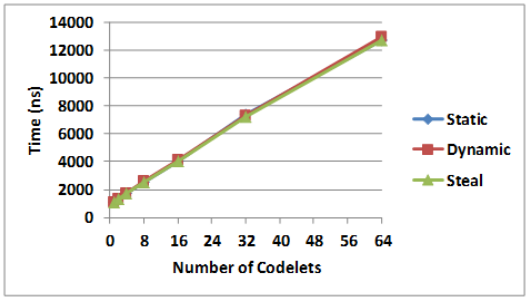
4.5.8.1 Mills

Figure 4.21(a) and 4.21(b) show the chain pattern while scaling the length of the chain. As clusters are added, the overhead increases. The scheduling policies do not factor into the performance. For this experiment we only use TP schedulers without CD schedulers. Since there is no parallelism, adding CD schedulers only adds overhead which is already explored in section 4.5.7. Figure 4.21(c) and 4.21(d) demonstrates the performance of the chain pattern while fully utilizing each cluster. Each scheduling policy demonstrates different overheads since each handles pushing and pulling codelets

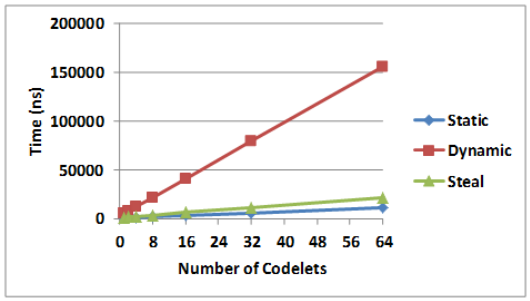
to the ready pool differently. The policies demonstrate the same trend as the fanout pattern in section 4.5.7.



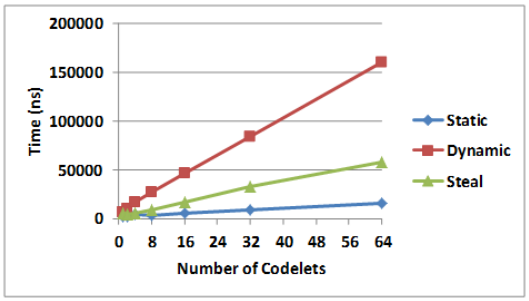
(a) Codelet chain of 8 codelets using 1 TP scheduler



(b) Codelet chain of 32 codelets using 1 TP scheduler



(c) 1 cluster running a codelet chain while scaling the number of codelets

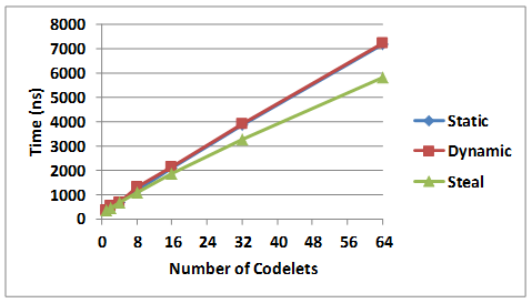


(d) The full system running a codelet chain while scaling the number of codelets

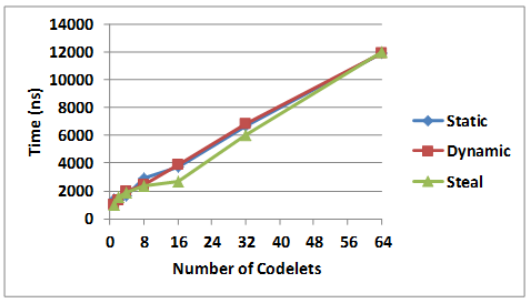
Figure 4.21: These graphs present the codelet chain pattern running on the Mills system.

4.5.8.2 Monica

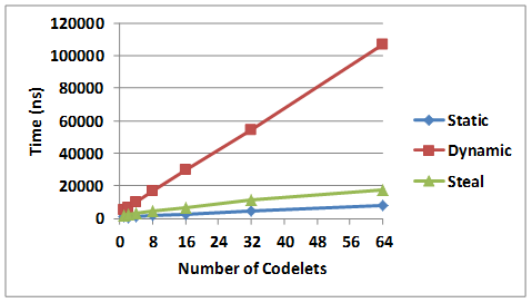
Figures 4.22(a) through 4.22(d) show the results of running the codelet chain pattern on the Monica system. In figure 4.22(a) we see the steal policy does slightly better as we scale the length of the chain. This trend does not stand as more clusters are used as seen in figure 4.22(b). Using the full system, we see similar patterns as Mills.



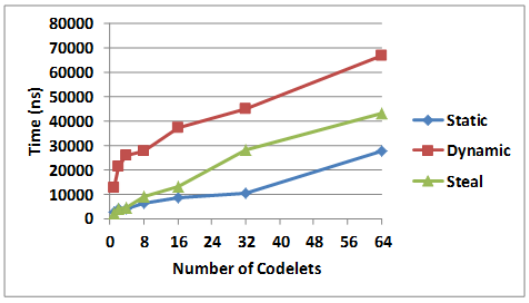
(a) Codelet chain using 1 TP scheduler while scaling the number of codelets



(b) Codelet chain using 4 TP scheduler while scaling the number of codelets



(c) 1 cluster running a codelet chain while scaling the number of codelets



(d) The full system running a codelet chain while scaling the number of codelets

Figure 4.22: These graphs present the codelet chain pattern running on the Monica system.

4.5.9 TP Chain

While the codelet chain pattern explores the cost of signaling codelets within a cluster, the TP chain pattern explores the cost of signaling codelets outside of a cluster. A TP chain consists of several TPs each with a single codelet. If the chain has reached the desired depth, the codelet signals its completion. Otherwise, the codelet invokes another TP. The static scheduling policy assigns the codelet to its TP scheduler.

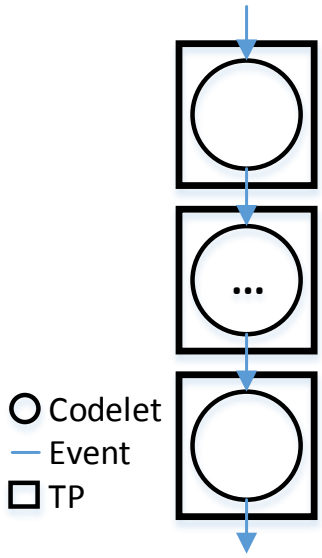
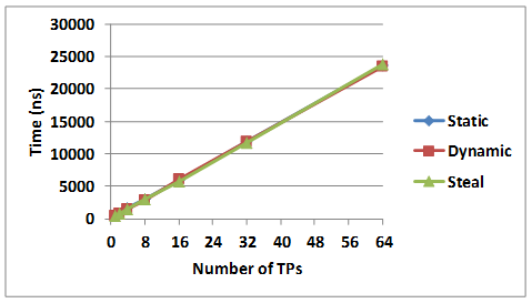


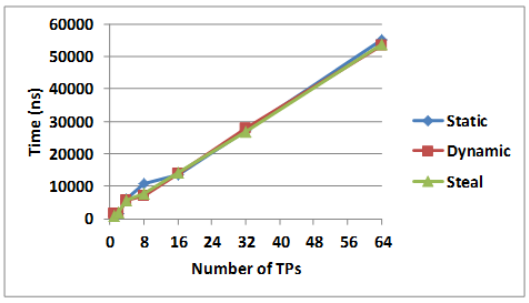
Figure 4.23: The TP chain pattern consists of TPs containing one codelet. The codelet is responsible for invoking another TP unless it is the last in the chain. If it is the last, the codelet signals the final codelet ending execution.

4.5.9.1 Mills

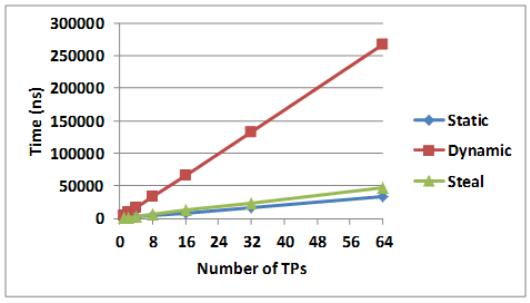
Figures 4.24(a) and 4.24(b) present the results of executing the TP chain pattern using a single TP scheduler and four TP schedulers respectively on the Mills cluster. In both graphs we see the results scale linearly with no difference between scheduling policies. Figures 4.24(c) and 4.24(d) show the TP chain running on one full cluster and four full clusters. The TP chain pattern exhibits the same pattern as the codelet chain at a higher cost. The increase in overhead can be attributed to two causes. The first is the additional overhead in using a TP rather than a single codelet as discussed in sections 4.5.4 and 4.5.5. The second is the out of cluster communication performed as TPs are load balanced.



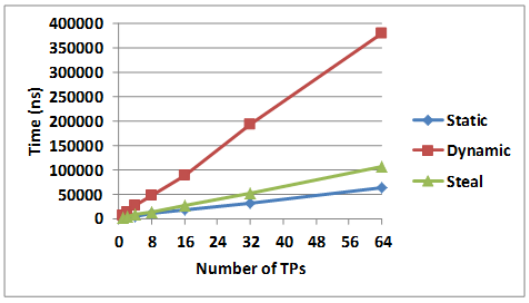
(a) TP chain using 1 TP scheduler while scaling the number of codelets



(b) TP chain using 4 TP scheduler while scaling the number of codelets



(c) 1 cluster running a TP chain while scaling the number of codelets

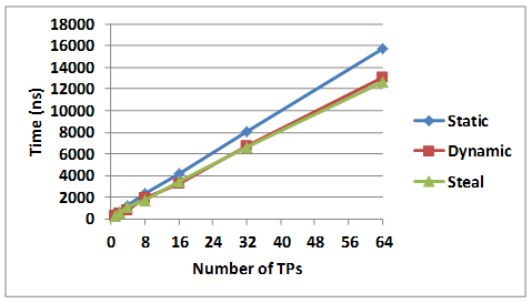


(d) The full system running a TP chain while scaling the number of codelets

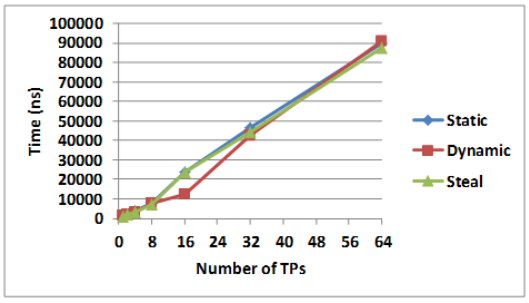
Figure 4.24: These graphs present the TP chain pattern running on the Mills system.

4.5.9.2 Monica

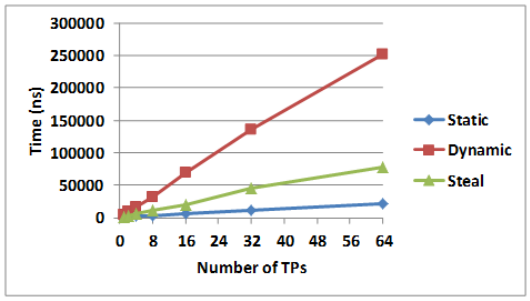
Figures 4.25(a) and 4.25(b) present the results of executing the TP chain on the Monica system. These two graphs are executed with only TP schedulers. The static scheduling policy performs worse than the other two policies when using only one TP schedulers, however all policies perform equally when using four TP schedulers. Figures 4.25(c) and 4.25(d) show the results of executing the TP chain pattern using an entire cluster and the full system.



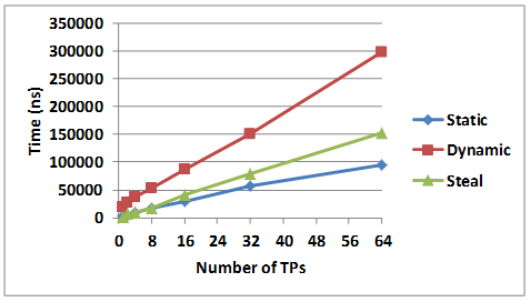
(a) TP chain using 1 TP scheduler while scaling the number of codelets



(b) TP chain using 4 TP scheduler while scaling the number of codelets



(c) 1 cluster running a TP chain while scaling the number of codelets



(d) The full system running a TP chain while scaling the number of codelets

Figure 4.25: These graphs present the TP chain pattern running on the Monica system.

4.5.10 Tree

The last pattern we explore is the tree pattern. The tree pattern begins with one TP which recursively invokes two new TPs. Each of those TPs invokes an additional two TPs until the desired depth is reached. Moreover we explore two version of the tree pattern. Leveraging the naming convention from [10] we present a strict and non-strict version of the tree pattern. TPs in the strict tree have two codelets, the first invokes the children TPs and the second is signaled once the children TPs have finished. The non-strict tree's initial TP has two codelets. All subsequent TPs in the tree have only one codelet. When the appropriate depth is reached the codelet signals the initials TP's second codelet rather than its immediate parent.

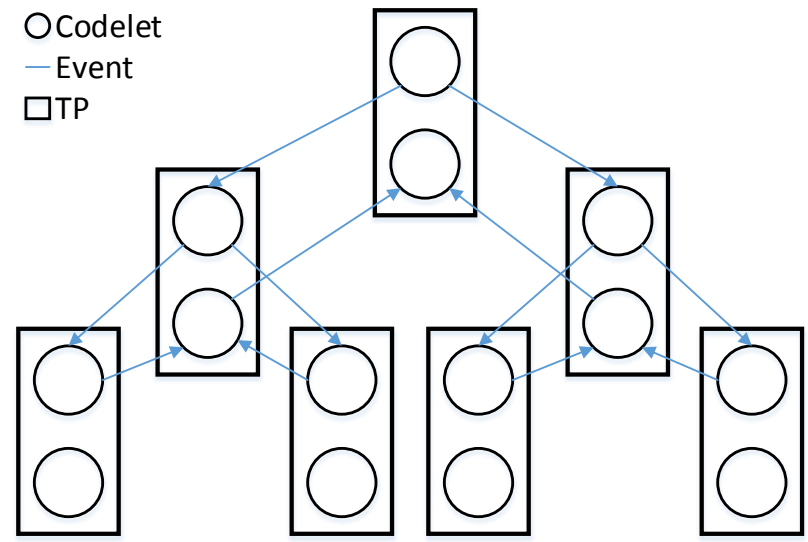


Figure 4.26: This tree pattern is fully strict. It is considered strict since every level other than the root TP returns to its invoking TP.

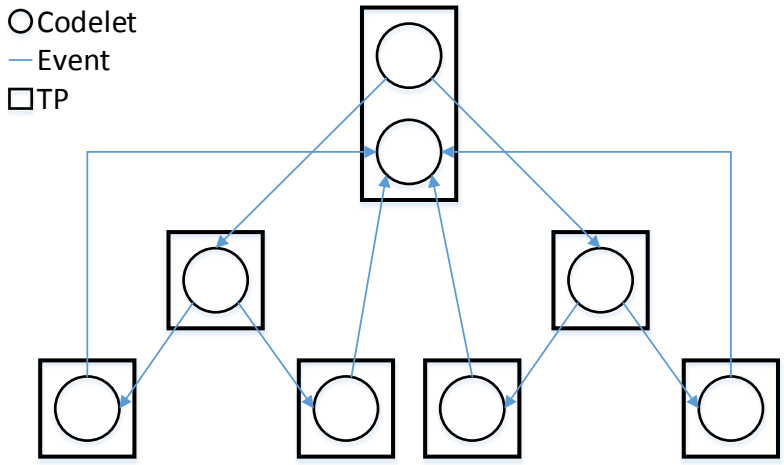
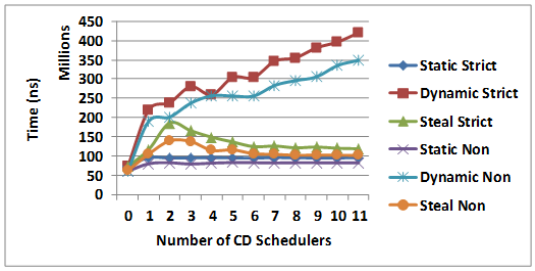


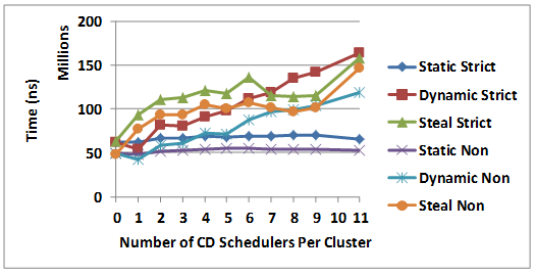
Figure 4.27: This tree pattern is non-strict. Rather than return to its immediate parent, a TP only signals the root TP upon completion.

4.5.10.1 Mills

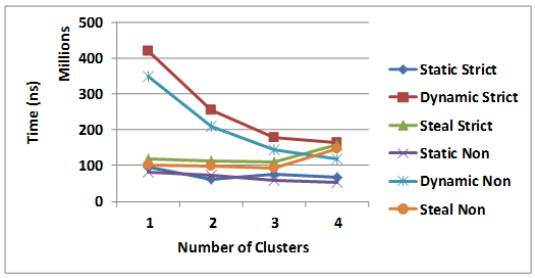
We present several views of the tree pattern using various configurations of the Mills system. Figure 4.29(a) presents a tree of depth 16 running on a single cluster. For this experiment we scale the number of CD schedulers. Once again we see the dynamic policy incur greater overhead as schedulers are added. The static policy suffer the least overhead since the codelets are schedule only on the TP schedulers (i.e. none of the CD schedulers run any codelets). The steal policy has greater contention on each schedulers RP when there are fewer CD schedulers. This diminishes as schedulers are added.



(a) Tree pattern using 1 TP scheduler and scaling the number of CD schedulers



(b) Tree pattern using 4 TP scheduler and scaling the number of CD schedulers



(c) Tree pattern scaling the number of fully utilized clusters

Figure 4.28: These graphs present the tree pattern of depth 16 running on the Mills system.

Figure 4.29(b) present the tree of depth 16 running using four cluster while scaling the number of CD schedulers per cluster. We see similar trends with the

exception of the steal policy performing worse than the dynamic policy while using less than half of each cluster.

In figure 4.29(c) we present a tree of depth 16 running on each cluster fully utilized. Here we notice both dynamic and static polices perform better as more clusters are added.

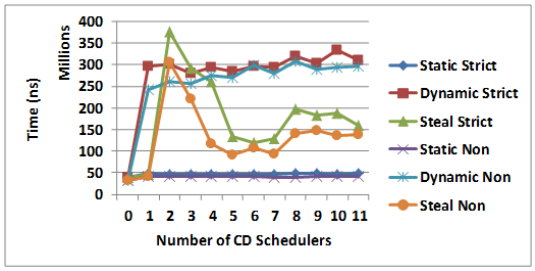
Lastly, we observe the difference between running the strict and non-strict trees. Table 4.4 present the average speedup non-strict demonstrates over the strict tree. This speedup originates from two places. The first is the non-strict tree only allocates half of the codelets (with the exception of the initial TP). The second is the signaling, scheduling, and running the extra codelets. Since each policy has different overheads in scheduling and running codelets, we see different speedups correlating the overheads of each policy.

Static	Dynamic	Steal
1.169	1.353	1.248

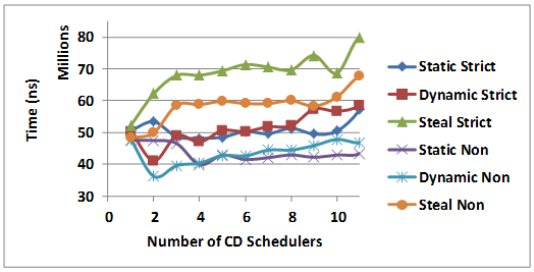
Table 4.4: This table shows the speedup of a non-strict tree over a strict tree running on the Mills system.

4.5.10.2 Monica

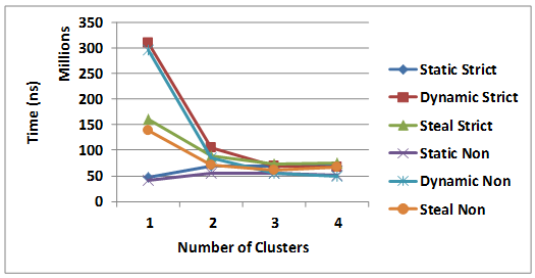
We present the same experiments as on before running on the Monica system. Figure 4.29(a) presents a tree of depth 16 running on a single cluster. On the Monica system we see the dynamic policy is much more stable. Contrarily, we see the steal policy exhibits an increased overhead once CD schedulers are added. Figure 4.29(b) presents a tree of depth 16 running using four clusters while scaling the number of CD schedulers. We see the steal policies perform the worst. Lastly, in figure 4.29(c) we present a tree of depth 16 while scaling the number of fully utilized clusters. Both the dynamic and steal policies perform better as clusters are added. The static polices perform relatively the same.



(a) Tree pattern using 1 TP scheduler and scaling the number of CD schedulers



(b) Tree pattern using 4 TP scheduler and scaling the number of CD schedulers



(c) Tree pattern scaling the number of fully utilized clusters

Figure 4.29: These graphs present the tree pattern of depth 16 running on the Monica system.

Table 4.5 present the speedup of non-strict trees over strict trees. Again we see the same patterns as presented for the Mills system.

Static	Dynamic	Steal
1.191	1.266	1.287

Table 4.5: This table shows the speedup of a non-strict tree over a strict tree running on the Monica system.

Chapter 5

CASE STUDIES

In the following chapter we evaluate DARTS running two different benchmarks to better understand the Codelet model. We attempt to provide a more complete view of DARTS' performance by using matrix multiplication and breadth first search. We continue to use the Mills system described in section 4.5.6.1. For all benchmarks, we will compare DARTS' results to an OpenMP version. Our objective is to provide a meaningful comparison between the DARTS and OpenMP runtime, thus each models' benchmarks are written similarly.

5.1 Matrix Multiply

We use Dense Square Matrix Multiplication (DGEMM) to observe DARTS' performance on a common compute-bound kernel. Moreover, many scientific applications rely heavily on linear algebra making DGEMM a meaningful case study. Regular kernels such as DGEMM typically perform well in OpenMP-like environments as work can be divided relatively uniformly.

In this study, we leverage a highly tuned serial DGEMM kernel as an optimized building block in our DARTS implementation. Figure 5.1 illustrates our decomposition of the DARTS version of DGEMM. We divide matrix A into rows and matrix B into columns producing a tile of results stored in matrix C. We leave the "inner tiling" to the tuned sequential kernel. The partitioning of the matrix is done by two loops, a parallel for all loop and a codelet for all loop. The parallel for loop divides matrix A into rows, while the codelet for all loop divides matrix B into columns. Moreover, each iteration of the parallel for loop copies the rows of matrix A into its TP bringing

the data local to the cluster. Each codelet in the codelet for all loop performs the multiplication locally and copies its tile into the C matrix.

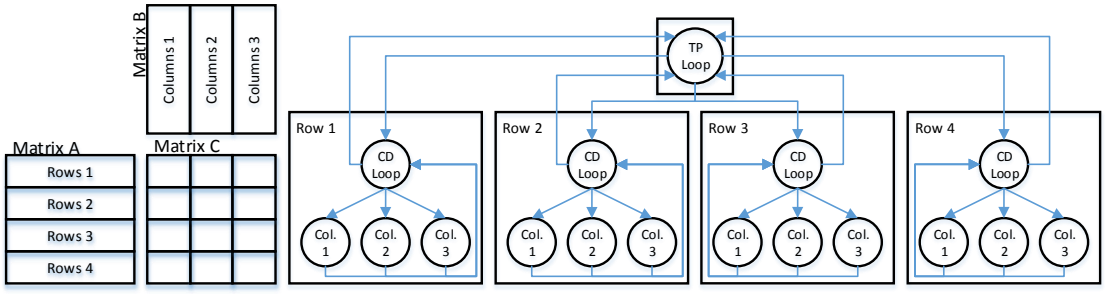


Figure 5.1: DARTS' DGEMM divides matrix A into rows and matrix B into columns. Rows of A are stored in a TP and shared by codelets within a codelet loop. Each codelet's result is a single tile of matrix C.

Since the Mills cluster is comprised of AMD cores, we use AMD's Core Math Library (ACML). The DARTS implementation uses ACML's tuned sequential kernel to compute each inner tile of the C matrix. We compare the DARTS results with ACML's parallel OpenMP DGEMM.

Figure 5.2 compares ACML's performance against DARTS' three policies all using the full system (48 cores). We present the absolute speedup relative to ACML's sequential performance. For matrices less than 500, ACML's OpenMP DGEMM performs on par with DARTS' static and dynamic policies. Once the matrices no longer fit in cache, ACML's performance drops and grows steadily afterwards. We see both the static and dynamic policy either outperform or tie the performance of ACML.

While figure 5.2 presents weak scaling for DGEMM, figures 5.3 and 5.4 present strong scaling for matrix sizes 1000 and 10000 respectively. DARTS static and dynamic policies scale linearly, while the steal policies drop off near the machines full capacity. ACML also scales linearly for matrix size 10000, however it flattens around 40 cores. We see drops in performance for DARTS from 4 to 5 cores and for ACML from 8 to 9 cores. This is due to the sharing of FP units between two cores. For ACML, the cores

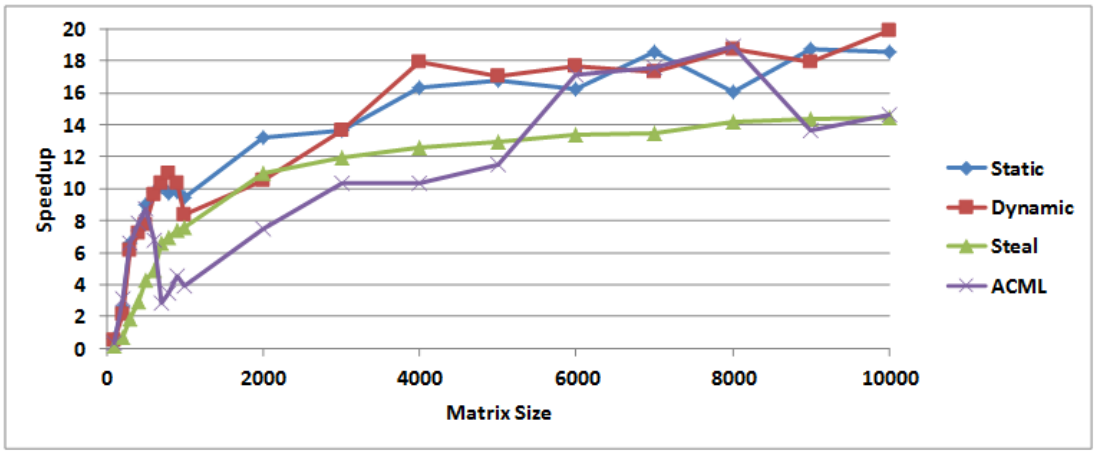


Figure 5.2: Weak scaling of DGEMM. We present the absolute speedup of DARTS and ACML's OpenMP while scaling the problem size (NxN).

are spread between the sockets requiring no sharing until more than 8 cores are used. DARTS drops sooner since the schedulers are pinned next to each other once there is more than one core per cluster. For the smaller problem size, ACML underperforms when utilizing half or more of the system. Table 5.1 summarizes DARTS speedup over the ACML OpenMP implementation.

	Static	Dynamic	Steal
Minimum	0.839	0.687	0.221
Average	1.558	1.535	1.044
Maximum	3.633	3.684	2.340

Table 5.1: DARTS speedup over ACML's OpenMP DGEMM

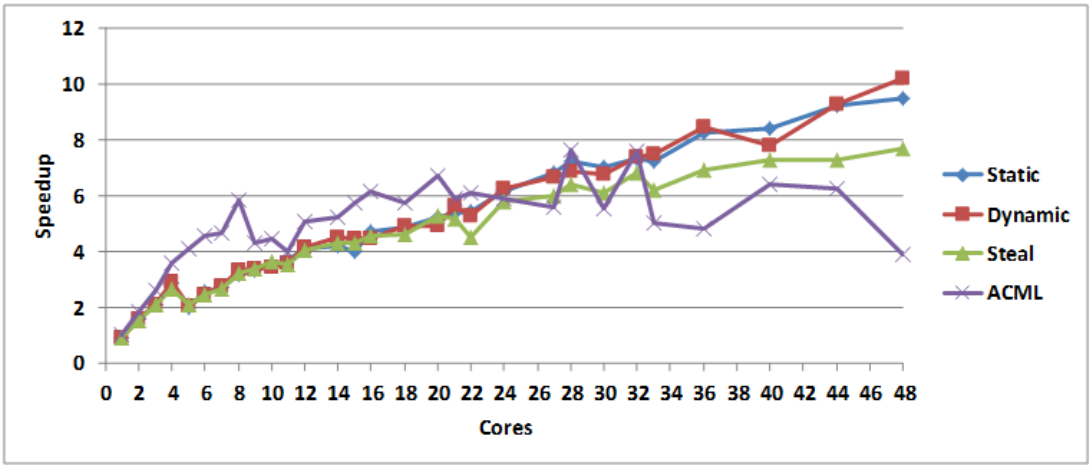


Figure 5.3: Strong scaling of DGEMM size 1000x1000. We present the absolute speedup of DARTS and ACML's OpenMP while scaling the number of cores.

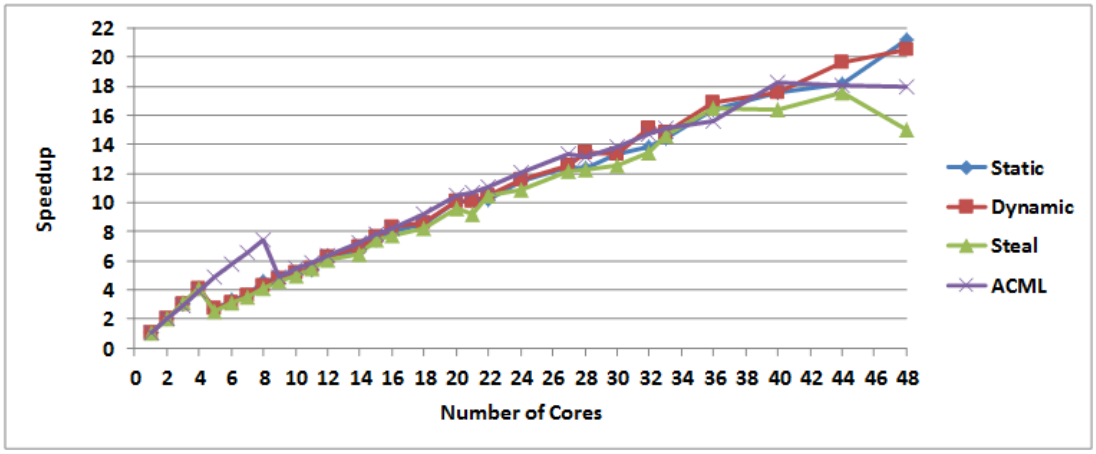


Figure 5.4: Strong scaling of DGEMM size 10000x10000. We present the absolute speedup of DARTS and ACML's OpenMP while scaling the number of cores.

As previously stated, OpenMP typically performs well for regular applications such as DGEMM. DARTS is capable of performing on par or better than ACML since tasks are collocated together in clusters leveraging DARTS' for loops. This enables codelets in a cluster to share rows of matrix A. The static and dynamic policies perform better than the steal policy since DGEMMs load can be more easily balanced.

5.2 Breadth First Search

To further evaluate DARTS, we used the Graph 500 parallel breadth-first search (BFS) algorithm[35]. BFS represents a class of irregular applications as the latencies of the memory accesses are dependent on the input data and subjected to NUMA effects. This leads to very unbalanced workloads, requiring sufficient load-balancing mechanism.

Graph 500's BFS algorithm performs an in order BFS search over an undirected graph outputting a spanning tree. The kernel begins with a single vertex in a graph. The vertex is "visited" by marking its children (neighboring nodes), and the children are placed in the search frontier. The search frontier is explored in phases. After all of the first vertex's children are enqueued, exploration of the new frontier is commenced. Each node in the frontier is visited, and if its children have not already been marked, they are added to the next phase's search frontier. These phases continue until all nodes in the graph have been visited. Parallelism is introduced into BFS by visiting nodes within a search frontier concurrently.

The OpenMP kernel distributes the nodes in a search frontier using a parallel loop. After exploring a single search frontier, the OpenMP threads enter a barrier before exploring the next frontier. By default, the reference implementation uses static scheduling. We only present results using static scheduling. Any experiments done using dynamic scheduling did not approach the static scheduling's performance.

The DARTS implementation uses a barrier-like approach similar to the reference implementation. The search frontier is distributed to one or more codelets, and a sink codelet is used to end each phase. We however, take advantage of the two-level

parallelism inherent in the codelet model. When the frontier is very small, we use a single codelet to process the frontier. As the frontier grows, we scale the parallelism using a codelet for all loop. This limits the parallelism to a single cluster, reducing the overhead of useless parallelism (parallelism which provides no gain in performance) and increasing data locality (of the search frontier). Once the frontier is large enough, we partition it into TPs using a parallel for all loop and again into codelets using a codelet for all loop. Figure 5.5 illustrates our strategy.

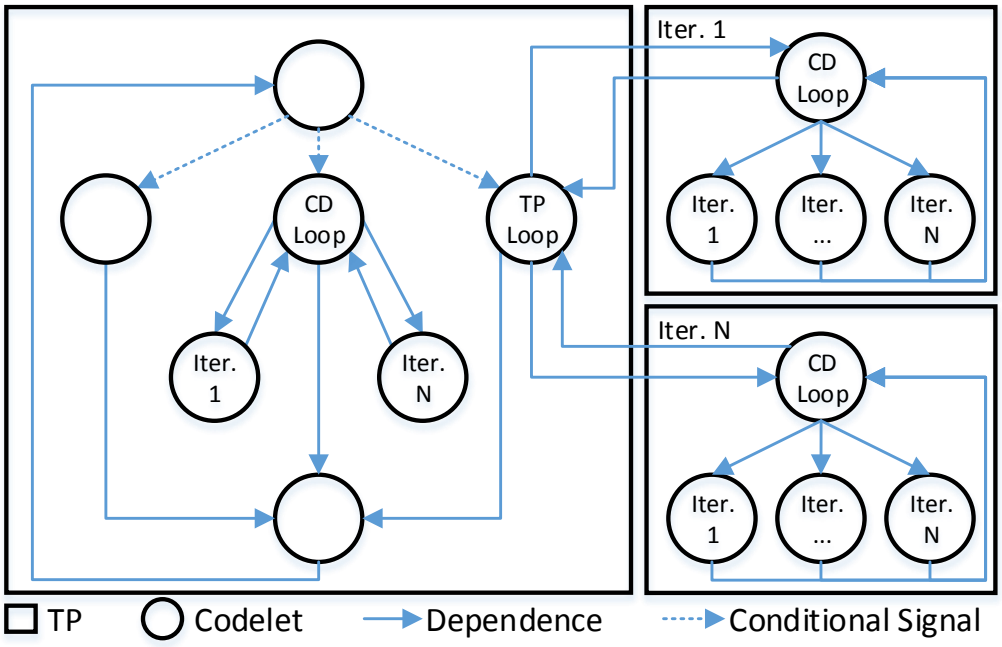


Figure 5.5: DARTS' BFS implementation partitions the search frontier based on its size. If the size is small a single codelet is used to process the frontier. A codelet loop is used to process a larger frontier. Nested TP and codelet loops are used to process the largest frontiers.

In the following sections we report the harmonic mean Traversed Edges Per Second (TEPS). Higher is better. To ensure a fair comparison, both DARTS and the reference implementation traverses the same graph. We use the RMAT graph generator

provided with the graph500 benchmark to produce a graph with 16 edges per node.

In figure 5.6 we present the weak scaling of graph500's BFS. We scale the degree of the graph from 14 to 26 while reporting TEPS. We see for smaller graph sizes both the static and dynamic policy clearly outperform the reference implementation. As the graph grows, the all three decrease in performance. The steal policy however, does not suffer in performance. This is due to the better load balancing possible with a larger amount of codelets. With more codelets, the static policy struggles to deal with the accumulating NUMA effects. The dynamic policy, suffers from the increased contention on RP. The steal policy mitigates both of these constraints as the graph grows. Table 5.2 summarizes the speedup of the DARTS policies over the reference implementation.

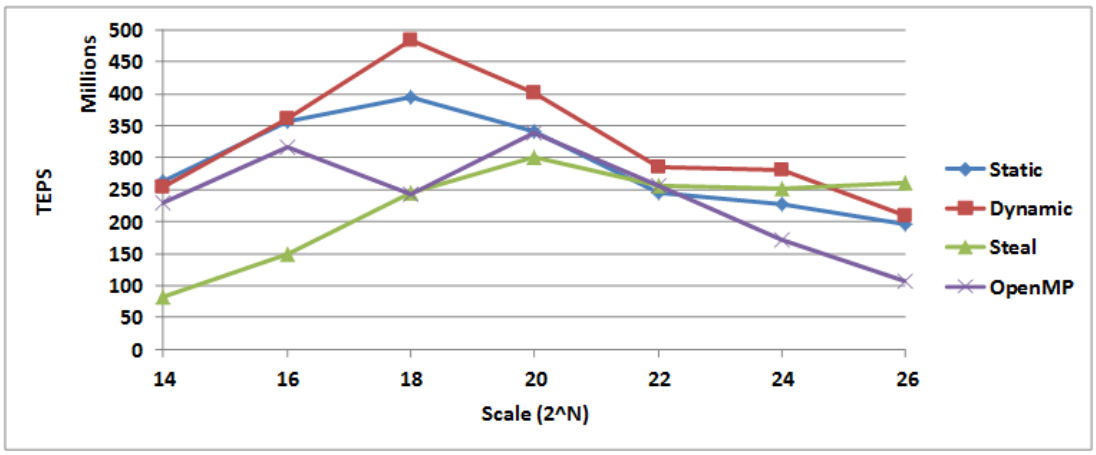


Figure 5.6: Weak scaling of graph500's BFS. We report TEPS (higher is better) while scaling the size of the graph.

Figure 5.7 presents the strong scaling for BFS. We see that all three of the DARTS policies scale better than the reference implementation. The work stealing policy is the closest to the reference implementation. Both flatten off around 16 cores. The static and dynamic policies do not however, and continue to scale to the full system.

	Static	Dynamic	Steal
Minimum	0.954	1.101	0.353
Average	1.289	1.447	1.091
Maximum	1.843	1.989	2.458

Table 5.2: DARTS' speedup over the OpenMP reference implementation

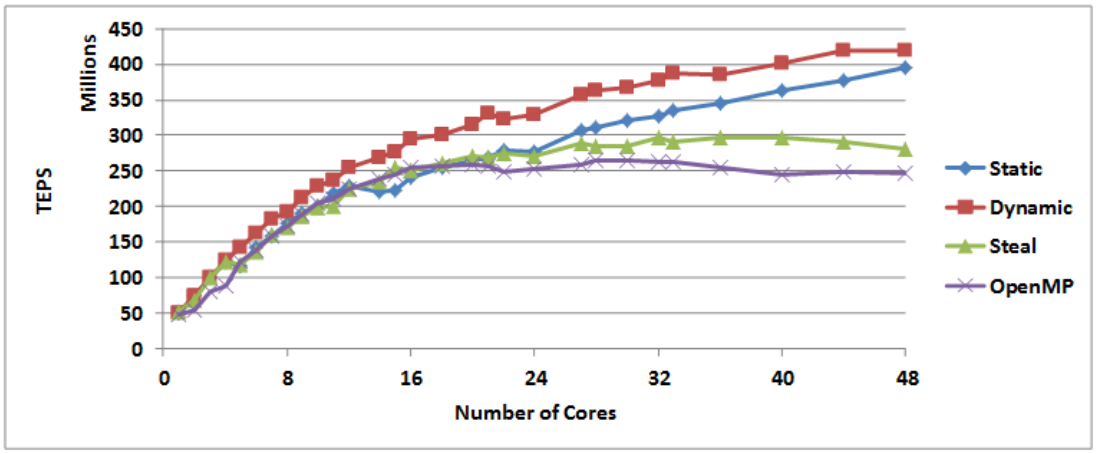


Figure 5.7: Strong scaling of graph500's BFS. We scale the number of cores processing a graph of size 2^{18}.

These results show the benefits of fine-grained parallelism for irregular applications. When there is a large amount of data to process, and significant load imbalance, DARTS shines. Moreover, we see the importance of having multiple scheduling policies. Leveraging the knowledge gained in section 4.5, we can better choose a policy dependent on the available parallelism understanding the overheads involved.

Chapter 6

RELATED WORK

As current execution models are struggling to scale, many have turned to dataflow ideas long since forgotten. The different approaches vary in granularity, programmability, and objectives. In the following chapter we explore several models each aiming for future large scale systems.

6.1 SWARM

The SWARM runtime [32] is a codelet based runtime developed by ET International. The runtime aims at increasing parallelism with low overhead while scaling for future architectures. SWARM is written in C, and can run on both shared and distributed memory systems. Moreover, SWARM is capable of leveraging a wide variety of accelerators like GPU or Intel's MIC architectures. SWARM's abstract machine model divides a system (or cluster) into nodes. Nodes are comprised of numerous CPU and accelerators along with some memory. A CPU may be further decomposed into execution units with threads, registers, and local memory. This abstract machine model is not unlike the Codelet model's abstract machine model.

6.1.1 SWARM's Threading Model

SWARM's threading model has several similarities to DARTS. The fundamental unit of work in SWARM is the codelet. A codelet is defined as a small piece of an application which can run to completion without blocking. In addition, it is capable of suspending and continuing execution if necessary. A codelet is comprised of the following:

- Run Fork - the work to execute furthering the application

- Cancel Fork - what to perform in case of an error
- Codelet Context - data (typically a struct) to be used by a codelet or group of codelets
- Input - a void pointer used to point to data passed to the codelet
- Chaining pointers - pointers used to invoke codelets similarly to functions.

Dependencies are not associated with codelets by default. Rather, they can be added to a codelet through a separate API call. Codelets are executed when the dependencies are met and satisfied through API calls. Codelets can also be executed through chaining. By providing a codelet with a pointer to another codelet and a context, codelets can be chained together executing one after another. These mechanisms (next codelet and next codelet context) are intended to promote code reuse and duplicate a pass by value functionality. Lastly, codelets in SWARM can be grouped together in some ad-hoc fashion similarly to DARTS' threaded procedure. Complexes typically have a context shared by codelets which may run serially or in parallel.

6.1.2 Locality and Scheduling

SWARM acknowledges the challenging problem of balancing parallelism with locality. To address this, SWARM employs a locale tree. The locale tree divides the physical hardware based on the natural boundaries found in the architecture. Each locale in the locale tree has a scheduler and allocator. Active codelets have the ability to change their locales while running.

The locale tree is used to balance work across the entire system. Only leaf locales, locales with no decedents, are capable of executing codelets. The other locals are used to distribute work in a work stealing manner. An idle scheduler checks with its parent scheduler to find work. If no work is available, work may be stolen from its parent's siblings.

SWARM differs from DARTS in its adherence to the Codelet model; rather, SWARM attempts to offer a greater degree of freedom. This can be seen by the decoupling of codelets, dependencies, and codelet complexes. The goal of DARTS is

to help evaluate the validity of hierarchical threading for future architectures. While there is no document providing insight to the overheads of SWARM, more API calls are required to generate the configuration of the Codelet execution model. Moreover, SWARM's scheduling policy is fixed. The burden of manipulating this behavior is left to the application programmer through the locale tree. DARTS provides multiple policies, and is designed modularly it order to promote the exploration of new runtime scheduling polices.

6.2 Open Community Runtime

The Open Community Runtime (OCR) is an event-driven framework developed by Rice University and Intel [43]. OCR applications are written in C and run on both shared and distributed memory systems. This framework leverages event-driven, fine-grain parallelism coupled with movable data blocks and introspection aimed at future systems.

An OCR application is comprised of Event-Driven Tasks (EDT). EDTs are similar to the codelets found in DARTS with one main exception. Each EDT has its own number of required event before it can be executed. Events are separate objects which must be created and linked to an EDT. Prior to and TP's execution, all associated events are satisfied with a data block. This differs from DARTS' synchronization slot, which employ a counter to keep track of the dependencies.

In order to permit EDTs to migrate across a system, OCR requires EDTs to use data blocks when performing computation. This constraint permits sophisticated introspection capable of load balance a system not only based on performance, but also energy and reliability constraints. Currently OCR only supports a standard work-stealing policy as it is still under development.

OCR while heavily inspired by the Codelet model differs in its lack of hierarchical threading (i.e. TPs). Currently Ed Ts do not have a mechanism to group task or assign them to a specific location. These mechanisms are useful for guaranteeing locality while still exposing parallelism.

6.3 Concurrent Collections

Concurrent Collections (CnC) [9, 11] is a framework centered on using higher level languages. The goal of CnC is to separate the concerns of domain experts and tuning experts. CnC runs on both shared and distributed memory systems leveraging multiple languages including C, C++, Java, .Net, and Haskell. The CnC framework is dataflow inspired. Applications are described in a graphical manner expressing their data and control dependencies (similarly to how the CDG is used to express programs in DARTS).

CnC has three main constructs, step collections, data collections, and tag collections which are used to statically describe an application. At runtime, dynamic instances of these collections are generated. The following describes each collection:

- Step Collection - Corresponds to a distinct instantiation of a serial computation (i.e. procedure) with unique inputs
- Data Collection - Holds data for specific steps to consume
- Tag Collection - Links specific data items to a particular step

Dependencies are expressed in a step by requiring a data with a specific tag. If a step is executing and it tries to access a tag that has not yet been produced, the step will block and wait for the tag. This is an important difference from codelets which are non-blocking. Despite this difference, it might be possible to use CnC as a high level language capable of generating codelets.

6.4 OmpSs and OpenStream

The OmpSs programming model [21] uses pragma based directives to augment C and C++ code to provide dataflow like parallelism. OmpSs combines two models, OpenMP and StarSs, to provide a neat and familiar programming model. This is aligned with OmpSs' primary goal of being productive without sacrificing performance.

The principle unit of work in the OmpSs model is a task. A task is considered an independent piece of code which can be run in parallel. Dependencies are created

between tasks by the in, out, and inout directives. These directives indicate which tasks depend on other tasks. Additional directives like taskwait can also be used for synchronization.

Similarly to OmpSs, the OpenStream programming model extends the OpenMP model to include dynamic dependent tasks. OpenStream also leverages the in and out directives to generate dependency graphs. OpenStream goes further than OmpSs however by adding streams. This directly differs from the atomicity that OmpSs' tasks demonstrate. In [38] the authors translate OmpSs to OpenStream and show its performance benefits.

These models are enviable for their familiar environments and clean design. However they lack the ability to describe locality. With future system scaling in size, it will be important for execution model to provide a means of expressing parallelism without losing the locality between tasks. This will be important for not only performance but also in reducing energy consumption.

One final important difference between OmpSs and DARTS is the atomicity of a task vs codelets. A task can only "signal" another task once it is completed its execution. A codelet may signal another codelet and continue execution. The Codelet model's method helps overlap computation at a finer granularity, an approach OpenStream has also adopted with their implementation of streams.

6.5 DFScala

DFScala [27] is a dataflow library written for the high level language Scala. Scala is a general purpose programming language boasting support for Object Oriented (OO) and functional programming as well as Java interoperability. The DFScala library permits the construction and execution of dataflow graphs.

The graph's nodes can range from instructions to entire functions. Furthermore, a node itself may be a sub-graph. Nodes are dynamically constructed during runtime, while the dependencies (arcs) are statically typed. Tokens for DFScala's dataflow

graphs can either be pushed by a producing node, or pulled. The pulling of tokens permits DFScala to leverage legacy code.

DFScala's approach to implementing dataflow graphs is very similar to DARTS'. Both rely of extending classes to create nodes in a graph. This also leads to a similar methodology in executing the graph. DARTS differs from DFScala in its use of hierarchical threading to exploit locality. However, it is conceivable that DFScala could be used to generate codelets especially with its ability to leverage sub-graphs.

6.6 Intel Threading Building Blocks

Intel Threading Building Blocks (TBB) [41] is a library intended to support scalable parallel programming. This library attempts to break away from the traditional parallel paradigm of threads. Rather it abstracts away threads and focuses on tasks. TBB relies heavily on work stealing to efficiently distribute tasks across a system. In addition to a tasking framework, TBB provides templates for scalable patterns and containers. The DARTS implementation even leverages the TBB queues to implement the RP and TPP.

Since TBB 4.0, flow graphs have been added to support both static and dynamic dependency graphs. While these graphs contribute dataflow semantics to the TBB framework, they do not have an underlying abstract machine model or means to control locality.

6.7 Charm++

Charm++ [30] is a parallel programming model developed over the past two decades at UIUC. The framework is heavily reliant on OO programming and leverages C++. Charm++'s objective is to provide a portable framework which scales based on OO principles, and can run on any system with an MPI installation.

A program written for the Charm++ framework is decomposed into C++ objects, primarily chares. A chare contains data, sends and receives messages, and executes a task when messages are received. Charm++ employs message-driven execution.

That is a Charm++ application is executed by chares sending asynchronous messages to other chars enabling parallel execution.

Chares can be organized into a collection. This collection is spread across the system running the contained chares on multiple cores. A simple collection is a chare array. The chare array provides a convenient means for the programmer to address multiple chares without any concern as to their physical location. Charm++'s runtime system is capable of migrating chares in the array to load balance the system based on several policies. Moreover, Charm++ provides other objects including sequential, concurrent, replicated, shared, and communication objects to aid in parallel execution.

DARTS and Charm++ both leverage OO programming, using similar approaches to providing an API. In addition, chares, codelets, and TPs are all implemented as objects, and can be inherited. The primary difference between Charm++ and DARTS is Charm++'s message driven execution. Contrary to DARTS event-driven execution, Charm++'s chares rely on receiving messages in order to begin. Furthermore, these messages are more expensive and force applications to exploit a more coarse-grained form of parallelism.

6.8 Cilk

The Cilk Language and runtime system [6] originally began at MIT under the direction of Charles Leiserson. Since its inception, Cilk technology has been acquired by Intel and has been include in the ICC compiler. Cilk extends both C and C++ to provide multi-threading for shared memory systems. The Cilk framework is most famous for popularizing work stealing as a load balancing scheme. The developers of Cilk were the first to provide tight boundaries on both time and space for work stealing.

The primary extensions added by Cilk are the spawn, sync, and for operations. The for operation is similar to an OpenMP parallel for loop. The spawn keyword is used to enable function calls to execute in parallel. When a function is "spawned," the current function is suspended and the new function begins execution on the same core. Differing from a serial execution, the suspended parent function may be "stolen" by

another core. The sync keyword is used to synchronize all spawns performed by the parent function to ensure correct execution.

While the original creators of Cilk were inspired by dataflow, Cilk exploits parallelism at the function level making arbitrary producer/consumers relations difficult. Cilk does provide important benefits including bounded time and space constraints, which DARTS is incapable of providing. The suspension of execution of the parent function to work on the child function enables Cilk to exploit the busy leaves property from [7]. The busy leaves property is critical in proving Cilk's bounds. DARTS differs since codelets are non-blocking and continue execution until completion, hence they do not demonstrate the busy leaves property.

One shortcoming of Cilk is its lack of locality aware directives. Unlike DARTS, work is balanced across the entire system which may hinder parallel tasks from sharing data effectively. This is an important aspect as future systems scale in size, and energy efficiency become a priority.

6.9 Freshbreeze

The Freshbreeze execution model [15, 16] is multi-threaded model designed for the Freshbreeze architecture. This architecture proposes a novel parallel system leveraging write-once memory organized in a single globally shared address space. All memory in the Freshbreeze system is divided into chunks of 1024 bits addressed by a 64 bit globally unique identifier. The threading model is an adaption of a fork/join model. A master thread spawns several slave threads each which execute some task. Upon completion, tasks join using join points. Join points may lead to the execution of some operations on data produced by the slaves (i.e. a reduction).

Freshbreeze is unique in its use of chunks to share data. When a task wants to share data, it writes the data to a chunk and seals the chunk. Once sealed a chunk cannot be modified. Each chunk has a reference count keeping track of the number of tasks with valid references to that chunk. Once no more references are held, the chunk is garbage collected.

The Freshbreeze model differs in its exploitation of parallelism. Freshbreeze explores the fork/join model with immutable data, while codelets extend dataflow semantics to include events. Moreover, Freshbreeze provides no mechanism for exploiting locality.

Chapter 7

CONCLUSION AND FUTURE WORK

New challenges have arisen due to the physical constrains in chip fabrication forcing future architectures to seek performance by becoming even more parallel. In order to scale to these architectures, we have presented the Codelet execution model. This model benefits from fine-grained, event-driven execution.

To further study the Codelet execution model, we provided a Codelet architecture model and DARTS, a codelet runtime system for shared memory systems. We provided a series of micro benchmarks to analyze the overheads associated with codelets and TPs in the DARTS runtime. These benchmarks provide insight aiding in application development when determining the size of a codelet and appropriate scheduling policy. Lastly, we presented two applications, DGEMM and graph500's BFS, comparing DARTS' implementation against OpenMP. With these two benchmarks we demonstrated of maximum of 3.684x improvement over the state of the art ACML DGEMM, and a 2.458x improvement over the graph500 reference implementation.

Going forward, we intend to evaluate more benchmarks to cement event-driven, fine-grain parallelism as the requirement for future execution models. To this end, we must further demonstrate the utility of events addressing energy and resiliency. Lastly, we plan to extend DARTS to run on distributed memory systems.

BIBLIOGRAPHY

[1] Saman Amarasinghe, Mary Hall, Richard Lethin, Keshav Pingali, Dan Quinlan, Vivek Sarkar, John Shalf, Robert Lucas, and Katherine Yelick. Ascr programming challenges for exascale computing. *Exascale Programming Challenges, Marina del Rey, CA, USA, July*, 2011.

[2] Arvind and K.P. Gostelow. The u-interpreter. *Computer*, 15(2):42–49, Feb 1982.

[3] Arvind and R.S. Nikhil. Executing a program on the mit tagged-token dataflow architecture. *Computers, IEEE Transactions on*, 39(3):300–318, 1990.

[4] T. Ungerer Arvind, L. Bic. Evolution of dataflow computers. In L. Bic J.-L. Gaudiot, editor, *Advance topics in data-flow computing*, pages 3–33. Prentice Hall, 1991.

[5] Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Jon Hiller, Sherman Karp, Stephen Keckler, Dean Klein, Robert Lucas, Mark Richards, Al Scarpelli, Steven Scott, Allan Snavely, Thomas Sterling, R. Stanley Williams, Katherine Yelick, Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Jon Hiller, Stephen Keckler, Dean Klein, Peter Kogge, R. Stanley Williams, and Katherine Yelick. Exascale computing study: Technology challenges in achieving exascale systems, 2008.

[6] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55 – 69, 1996.

[7] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, Sep 1999.

[8] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst. hwloc: A generic framework for managing hardware affinities in hpc applications. In *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*, pages 180 –186, feb. 2010.

[9] Zoran Budimlić, Michael Burke, Vincent Cavé, Kathleen Knobe, Geoff Lowney, Ryan Newton, Jens Palsberg, David Peixotto, Vivek Sarkar, Frank Schlimbach, et al. Concurrent collections. *Scientific Programming*, 18(3):203–217, 2010.

[10] Vincent Cavé, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. Habanero-java: The new adventures of old x10. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, PPPJ '11, pages 51–61, New York, NY, USA, 2011. ACM.

[11] A. Chandramowlishwaran, K. Knobe, and R. Vuduc. Performance evaluation of concurrent collections on high-performance multicore computing systems. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12, April 2010.

[12] S. Chatterjee, S. Tasrlar, Z. Budimlic, V. Cave, M. Chabbi, M. Grossman, V. Sarkar, and Yonghong Yan. Integrating asynchronous task parallelism with mpi. In *Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 712–725, May 2013.

[13] A.L. Davis and R.M. Keller. Data flow program graphs. *Computer*, 15(2):26–41, 1982.

[14] J. B. Dennis and G. R. Gao. An efficient pipelined dataflow processor architecture. In *Proceedings of the 1988 ACM/IEEE Conference on Supercomputing*, Supercomputing '88, pages 368–373, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.

[15] Jack B. Dennis. Fresh breeze: A multiprocessor chip architecture guided by modular programming principles. *SIGARCH Comput. Archit. News*, 31(1):7–15, mar 2003.

[16] Jack B Dennis. The fresh breeze model of thread execution. In *Workshop on programming models for ubiquitous parallelism. IEEE Comput Soc, Los Alamitos. Published with PACT-2006*, 2006.

[17] Jack B. Dennis, Arvind, Guang R. Gao, Xiaoming Li, and Lian-Ping Wang. Architecture and programming model for high performance interactive computation. Technical Memo – Available on request, April 2014.

[18] Jack B. Dennis and David P. Misunas. A preliminary architecture for a basic data-flow processor. *SIGARCH Comput. Archit. News*, 3(4):126–132, Dec 1974.

[19] J.B. Dennis, J.B. Fosseen, and J.P. Linderman. Data flow schemas. In Andrei Ershov and Valery A. Nepomniaschy, editors, *International Symposium on Theoretical Programming*, volume 5 of *Lecture Notes in Computer Science*, pages 187–216. Springer Berlin Heidelberg, 1974.

[20] J.B. Dennis, G.R. Gao, and V. Sarkar. Determinacy and repeatability of parallel program schemata. In *Data-Flow Execution Models for Extreme Scale Computing (DFM), 2012*, pages 1–9, Sept 2012.

[21] Alejandro Duran, Eduard Ayguad, Rosa M. Badia, Jess Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. Ompss: A proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21(02):173–193, 2011.

[22] M. Flynn. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, C-21(9):948–960, 1972.

[23] G.R. Gao, H.H.J. Hum, and Y.-B. Wong. Parallel function invocation in a dynamic argument-fetching dataflow architecture. In *Databases, Parallel Architectures and Their Applications,. PARBASE-90, International Conference on*, pages 112–116, 1990.

[24] Guang R. Gao and Vivek Sarkar. Location consistency-a new memory model and cache consistency protocol. *IEEE Trans. Comput.*, 49(8):798–813, August 2000.

[25] Guang R. Gao, Joshua Suetterlein, and Stephane Zuckerman. Toward an Execution Model for Extreme-Scale Systems - Runnemede and Beyond. Technical Memo – Available on request, April 2011.

[26] Elkin Garcia, Daniel Orozco, Rishi Khan, Ioannis Venetis, Kelly Livingston, and Guang R. Gao. Dynamic percolation: A case of study on the shortcomings of traditional optimization in many-core architectures. In *Proceedings of 2012 ACM International Conference on Computer Frontiers (CF 2012)*, pages 245–248, Cagliari, Italy, May 2012. ACM.

[27] D. Goodman, S. Khan, C. Seaton, Y. Guskov, B. Khan, M. Lujan, and I. Watson. Dfscala: High level dataflow support for scala. In *Data-Flow Execution Models for Extreme Scale Computing (DFM), 2012*, pages 18–26, Sept 2012.

[28] Laurie Hendren, Xinan Tang, Yingchun Zhu, Guang Gao, Xun Xue, Haiying Cai, and Pierre Ouellet. Compiling C for the EARTH Multithreaded Architecture. In *International Journal of Parallel Programming*, pages 12–23. IEEE Computer Society Press, 1996.

[29] Herbert H. J. Hum, Olivier Maquelin, Kevin B. Theobald, Xinmin Tian, Guang R. Gao, and Laurie J. Hendren. A study of the EARTH-MANNA multithreaded system. *Int. J. Parallel Program.*, 24(4):319–348, Aug 1996.

[30] Laxmikant V. Kale and Sanjeev Krishnan. Charm++: A portable concurrent object oriented system based on c++. In *Proceedings of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '93, pages 91–108, New York, NY, USA, 1993. ACM.

[31] K.M. Kavi, B.P. Buckles, and U. Narayan Bhat. A formal definition of data flow graph models. *Computers, IEEE Transactions on*, C-35(11):940–948, 1986.

[32] Christopher Lauderdale and Rishi Khan. Towards a codelet-based runtime for exascale computing: Position paper. In *Proceedings of the 2Nd International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, EXADAPT '12, pages 21–26, New York, NY, USA, 2012. ACM.

[33] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. Nvidia tesla: A unified graphics and computing architecture. *Micro, IEEE*, 28(2):39–55, 2008.

[34] C. McCurdy and J. Vetter. Memphis: Finding and fixing numa-related performance problems on multi-core platforms. In *Performance Analysis of Systems Software (ISPASS), 2010 IEEE International Symposium on*, pages 87–96, 2010.

[35] Richard C Murphy, Kyle B Wheeler, Brian W Barrett, and Jim Ang. Introducing the graph 500. *Cray Users Group (CUG)*, 2010.

[36] Gregory M. Papadopoulos and David E. Culler. Monsoon: An explicit token-store architecture. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, ISCA '90, pages 82–91, New York, NY, USA, 1990. ACM.

[37] Suhas S. Patil. Record of the project mac conference on concurrent systems and parallel computation. chapter Closure Properties of Interconnections of Determinate Systems, pages 107–116. ACM, New York, NY, USA, 1970.

[38] Antoniu Pop and Albert Cohen. Openstream: Expressiveness and data-flow compilation of openmp streaming programs. *ACM Trans. Archit. Code Optim.*, 9(4):53:1–53:25, January 2013.

[39] Marc Prache, Herv Jourdren, and Raymond Namyst. Mpc: A unified parallel runtime for clusters of numa machines. In Emilio Luque, Toms Margalef, and Domingo Bentez, editors, *Euro-Par 2008 Parallel Processing*, volume 5168 of *Lecture Notes in Computer Science*, pages 78–88. Springer Berlin Heidelberg, 2008.

[40] Thomas Rauber and Gudula Rünger. *Parallel programming: For multicore and cluster systems*. Springer Science & Business, 2013.

[41] James Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. O'Reilly Media, Inc., 2007.

[42] Vivek Sarkar, William Harrod, and Allan E Snavely. Software challenges in extreme scale systems. *Journal of Physics: Conference Series*, 180(1):012045, 2009.

[43] Vivek Sarkar, Rob Knauerhase, and Rich Lethin. The open community runtime framework for exascale systems, Nov 2013.

[44] Joshua Suettlerlein, Stphane Zuckerman, and GuangR. Gao. An implementation of the codelet model. In Felix Wolf, Bernd Mohr, and Dieter Mey, editors, *Euro-Par 2013 Parallel Processing*, volume 8097 of *Lecture Notes in Computer Science*, pages 633–644. Springer Berlin Heidelberg, 2013.

[45] Kevin Bryan Theobald. *EARTH: an efficient architecture for running threads*. PhD thesis, McGill University, Montreal, Que., Canada, Canada, May 1999. AAINQ50269.

[46] Jurij Šilc, Borut Robič, and Theo Ungerer. Progress in computer research. chapter Asynchrony in Parallel Computing: From Dataflow to Multithreading, pages 1–33. Nova Science Publishers, Inc., Commack, NY, USA, 2001.

[47] L. Verdoscia and R. Vaccaro. A high-level dataflow system. *Computing*, 60(4):285–305, 1998.

[48] Ian Watson and John Gurd. A prototype data flow computer with token labelling. *Managing Requirements Knowledge, International Workshop on*, page 623, 1979.

[49] Haitao Wei, Guang R. Gao, Weiwei Zhang, and Junqing Yu. Costream: A dataflow programming language and compiler for multi-core architecture. In *Data-Flow Execution Models for Extreme Scale Computing (DFM), 2013*, Sept 2013.

[50] Stéphane Zuckerman, Joshua Suetterlein, Rob Knauerhase, and Guang R. Gao. Using a "codelet" program execution model for exascale machines: Position paper. In *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, EXADAPT '11, pages 64–69, New York, NY, USA, 2011. ACM.