Speculative Execution in a Peer-to-Peer Run-Time System

by Sean Halle

Abstract

An economically attractive infrastructure for parallel software is important for the continuing health of the computer industry. We have previously suggested the CodeTime platform as a blueprint for such an infrastructure. One element of this platform is a virtual server, which defines a computation model. Implementing this computation model requires some form of run-time system.

In this paper, we present a peer-to-peer run-time system for the CodeTime platform.

The run-time system is hierarchical, to allow scaling. It's overlay network is organised into a tree-graph with all children of a single node fully interconnected. A simple analytic model suggests that the number of control messages per peer increases logarithmically with the number of computing nodes in the system.

The control protocol is based upon the principle of speculative execution to increase efficiency. Each child of a node independently calculates the task of every other child in its group. In this way, both sender and receiver are expecting communication without exchanging any control messages, in particular no synchronization protocol is used in this system. Each peer in a group maintains identical state by sending a notice, when it completes a task, to each other member of the group. Thus, actions may be taken by a peer optimistically, in the expectation that its state for the other peers is correct. If a peer receives an unexpected request to receive data, it knows that a difference exists between its state and the sender's, and initiates a resolution protocol.

We discuss the implementation of this run-time system and give results showing its correct operation.

1 Introduction.

The generation currently under development for all four major lines of microprocessor are either multi-core, multi-threaded, or both. The number of threads on a chip is expected to double every 18 months. Therefore, future applications must be flexible in the number of hardware threads they efficiently make use of. Developing applications with this property has proven problematic.

The economic history of the computer industry has shown that the ability to run old software with reasonable efficiency on new machines is a powerful force. The continued existence of the x86 instruction set is testament to this. Internally, the Pentium processors implement a custom RISC-style instruction set, yet, this inner core is still surrounded by a superfluous shell that translates x86 instructions into this internal RISC format. Many have estimated that processing could be improved by 50% or more by eliminating this shell. Yet it persists. The reason lies in the economics of software distribution. Customers do not want to buy and re-install new software each time they upgrade their machine.

For parallel software, an equivalent of the x86 instruction set has not been found. Abstract machines have been suggested, but have proven too inefficient in practice.

The CodeTime platform introduces a computation model which may have the desired properties. It modifies and extends the dataflow computation model by removing the model's implied constraints on the order of operations, and replacing them with a programmer-defined set of constraints. The rest of the platform is put in place to support this extension and derive the benefits enabled by this extension.

2 Background

The CodeTime platform consists of a virtual server definition, a family of source languages, and a development environment which supports and implements some of the source-language features.

The virtual server appears to users as a single machine, yet may have any number of physical processing nodes behind this interface. The "processor" of this machine is a mutable circuit. Code compiled from source specifies a particular set of circuit elements and a wiring among these elements.

A virtual server is implemented separately on each hardware platform. One of the tasks of implementing a virtual server instance is making some way for the hardware to behave as the specified circuit. One way to do this is to split the behavior into static and dynamic.

The static behavior is compiled by a hardware-platform-specific back-end compiler. The output of this compiler is a set of modules of machine-code, ready to execute. Embedded into this machine code are calls to something which implements the dynamic behavior of the circuit.

The dynamic behavior of the circuit includes choosing which physical processor should execute each module of compiled code. When a module has completed, another choice must be made for which processor gets those results, and which module to apply to them. This dynamic behavior is implemented by a run-time system.

To adapt to different numbers and types of physical processors, the platform must provide a means to change the granularity of a computation. One way to adapt is to give the run-time system a way to dynamically divide data into sizes which fit best on the hardware. One way to accomplish this is by using a divide-compute-undivide pattern, and mapping all application code onto this pattern.

The semantics of the source language and the computation model allow the back-end compiler to accomplish this mapping. The output from the compiler includes a machine-code module which divides data into a set of chunks. The output also includes a machine-code module which takes a number of result-chunks and reassembles them.

The semantics also support combining several circuit elements into a single conglomerate element. The circuit elements are called code-units, and the conglomerate elements are called combined-units or dashed-units. The back-end compiler chooses which circuit elements to combine according to the details of the hardware. Each combined-unit is compiled as a single executable module.

The run-time system thereby receives a number of executable modules, including one which divides data, one which "undivides" data, and number which perform steps in the computation. At the end of the execution of each module, control passes back to the run-time system.

The platform supports profile information for these modules, which gives the run-time the ability to estimate how long a given size chunk will take to pass through a given executable module. This is useful when making the decisions about how large each chunk should be and which processor it should run on.

This paper gives the details of the implementation of one kind of run-time system, for hardware made up of a number of processors interconnected by a network. High-end computers, networks of workstations, and grid computing are classic examples of this hardware model. It might also be useful for Shared-memory servers, such as Sun enterprise class machines, which fit this hardware model when the memory hierarchy is perceived by the compiler and run-time as a very flexible network. Multicore processors also fit this framework, and will progressively fit it better over the next ten years, which is the anticipated timespan over which platforms such as CodeTime will become adopted.

Section 3 discusses related work, section 4 gives an overview of the run-time system and communication protocol, section 5 discusses the details of this implementation, section 6 gives screen shots produced by this implementation correctly completing one divide-compute-undivide sequence, section 7 suggests improvements to the run-time system, and section 8 restates some important points from the paper.

3 Related Work

(Note to reader, the author has run out of time. This section is left blank, and the bibliography has not been gathered. These will both be corrected over the coming months as this paper and the work are readied for submission for publication.)

4 Overview

The run-time system is a symmetric peer-to-peer system, by which is meant that no services are centralized. All functions of the system may be performed by any of the processing nodes at a given moment. The term peer has meaning, potentially confusingly, in two different senses. In the hardware sense, a peer is a processing-node. In the software sense, a peer is a virtual entity. One or multiple of these may be assigned to a single processing-node. In this paper, processing-nodes are always referred to as processing-nodes, and the term peer always means the virtual-software-entity.

Peers are organized into an overlay network. The overlay network is a tree-graph. Each node in the tree-graph is a peer, and each child of a node is fully connected to each other child of that node. The peers are mapped onto processing-nodes. A one-to-one relationship exists between each processing-node and a leaf-peer. A given processing-node may also have a peer which is a member of any level(s) of the hierarchy above the leaves. If a processing-node has multiple peers, no particular relationship in the tree must exist among that processing-node's peers.

The peers each keep the status of every other peer in its peer-group (the peers it is fully-connected to). Each time a peer completes some unit of work, it sends a message to every other member of its peer-group. When a peer receives a completion message, it updates its status table and decides what action, if any, to take as a result.

This way, all the peers may calculate the action which each other peer should take as a result of every completion message. When a peer receives a request for data-transfer from another, it calculates whether it agrees that this transfer should take place. If it does not, it initiates a discrepency-resolution process.

A computation begins when the root peer is handed a set of data to compute the result of. The root chooses one of its children to perform the un-division process. The root broadcasts this choice, along with meta-information about the data, to each child-peer.

All the peers in the group then calculate how to divide up the data, and which piece they should take. Each peer then requests their piece from the data-owner, who could be anywhere in the network. The data-owner responds with that piece of data. The peer then applies the first code-unit to its piece of data. When done, that peer tells all the others in the group, and the parent. All the peers then decide whether this piece allows one of them to collect an input set, and which peer should get it.

If it is possible to gather a complete input set, then the one which should goes ahead and requests the result pieces from the others. When it has them all, it applies the appropriate combined-unit to that input-set. When done, it notifies the other peers, which all then repeat the process of deciding whether an input set can be formed and which peer should get it.

When a result comes out of the last combined-unit, the peer which produced it sends out a completion message, then sends the result to the designated un-divider peer. The un-divider peer applies the un-divide-unit to the result. When done, it sends out a completion message, then checks whether all the results have been gathered.

When all the results have been gathered into the un-divider, the un-divider peer sends the combined-result to the parent-peer. The other peers in the group will see the completion message and deduce that the parent has been given the result.

That parent then informs its peer-group that a result is complete, and things proceed on this level in the same fashion as they did on the leaf-level. The one difference is that the peers above the leaf-level may only delegate computation of combined-units which contain their own divide-compute-undivide pattern. Not all possible combined-units will have this. Thus, the above-peers have fewer control-parallelism choices available to them. For this reason, the combined-units which the compiler makes for the higher-level peers will usually be different than the combined-units it makes for the leaf-level peers. This maximizes the available control-parallelism at the leaf level.

When the root receives the result from the un-divider it delegated, it in turn hands that result back to the requestor. The details of interaction between the root of the run-time system and the rest of the virtual-server are available on the SourceForge site: http://codetime.sourceforge.net

5 Implementation

5.1 Composition of a peer

Each peer is composed of four pieces:

- A Server, which accepts all messages to the peer
- A Client, which sends all messages from the peer
- A Scheduler, which tracks the status of all the peers in the group and decides what action each of them, including itself, sould take in response to each incoming message.
- A Worker, which applies combined-units to sets of data. The scheduler collects each set of data, then hands that set to the worker, along with which combined-unit (compiler-generated executable module) to apply.

The worker has no persistent state. What the scheduler gives the worker contains everything needed to perform the computation. This is an important property. It is at the heart of why the run-time is able to divide data into smaller pieces so easily. This property is enabled by the central idea the CodeTime platform is built around, namely the addition of coordination-constraints to dataflow.

A peer also contains three blocking queues to connect the pieces. The queues are used one-directionally. One goes from the scheduler to the client, one from scheduler to worker, and one from both worker and server to the scheduler.

5.1.1 The Server

The server is a standard TCP/IP server, which listens on a port for clients attempting to connect. This implementation has all the peers on the same processing-node, so they all have "localhost" as their host name. Different peers are identified by the port they listen on.

The server and client communicate serialized RTMsg objects. When one is received, the server places it on the queue to the scheduler.

5.1.2 The Client

The client takes message objects off the queue from the scheduler, takes the "to" port out of it, establishes a connection to the server listening on that port, and sends the object.

5.1.3 The Scheduler

The scheduler takes messages off the queue going to it and decides what action to take as a result. It implements all of the communication protocol logic, and all of the decision making about which peer should perform which operation. This decision making which assigns actions to peers is a scheduling algorithm.

The communication protocol is heart of what defines this run-time system.

The scheduling algorithm, however, is a separable piece. A wide variety of different algorithms may be plugged in. This implementation uses an extremely simple algorithm which simply divides the incoming data evenly among all the peers in the group. When a peer acts as a parent, it randomly assigns one of the peers below it to be the undivider.

Thus, this scheduler does not take into account the status of the other peers in the group, and so the discrepency resolution process is not yet implemented.

The scheduler picks which of the combined-units should be applied to a piece of data. In the version implemented, only three combined units exist: the divider, the "main" work-unit, and the undivider. The scheduler picks the divider in response to a "here's a chunk to divide up" (chainReqDataTakeMsgFromAbove) message from the parent. It picks the main work unit in response to a "here's the piece you requested"(respDataTakeMsgFromAny) from the owner of the chunk. It picks the undivider in response to a "here's a result to be undivided"(respDataTakeMsgFromSame) from one of the peers in the group.

5.1.4 The Worker

The worker models the compiler-generated executable modules by having a method for each of the three units. The scheduler puts into the message it gives to the worker which of the units should be applied. The worker then executes the appropriate method, giving it the data information from the incoming message (reqWorkMsgFromSelf). The data is modelled as simply an integer, stating the size of the data. When the method completes, it hands back the result-size, which the worker puts into a response message (respWorkMsgFromSelf) that it then places into the queue going back to the scheduler.

5.2 The communication protocol

The communication between run-times is also one-directional. Clients inside the peer objects only send messages, servers only receive. This organisation mirrors the asynchronous nature of the computation model. This asynchrony of the computation model is also an important property. It is a reason that no synchronization operations are needed among peers, making it profitable to take actions speculatively. This property is enabled, also, by the coordination-extension to dataflow that the CodeTime platform is built around.

5.2.1 The sequence of messages

The sequence goes:

→ Parent sends: ReqChainDataTakeMsgFromAbove To: all peers in the group below it

- Each peer independently calculates how to divide up the data. The msg contains meta-info about the data, but not the actual data itself. In this impl, the data size is just divided by the number of peers.

→ In response, Peers each send: ReqChainDataGiveMsgFromAny To: the owner of the data

- In this impl, no data actually moves. In a real impl, all requests would go back through the chain. One of the higher-level peers in the chain would track the requests, to ensure that no requests overlap, and all the data in the original piece is requested.

→ In response, data owner (the parent in this case) sends: RespDataTakeMsgFromAny To: requestor

- The data-owner just makes a new message with the same size. In the real system, it would place the actual data into this response message.

→ In resp, each peer sends: ReqWorkMsgFromSelf To: worker

- The worker applies the combined-unit chosen by the scheduler to the data specified in the message.

→ In resp, worker sends: RespWorkMsgFromSelf To: scheduler

- In this impl, it just sends back the size of the data.

→ In resp, peer sends: InfoComplMsgFromSame To: each other peer in group

- When a peer receives this message, it only updates its copy of the status of the peers in the group (including its own status). The status will be used in real implementations to independently decide how to divide up data, and which peer should receive each completed input-set.

→ The peer also sends: InfoComplMsgFromBelow To: the parent

- The parent also keeps the status of every peer in its child-group. It uses this to pick an undivider. (Other method might be letting the group pick the undivider). It also keeps the parent informed about the progress of the computation. In this impl, no status is kept, the parent picks an undivider at random.

→ The peer also sends: RespDataTakeMsgFromSame To: the undivider (if peer is not undiv)

- When a peer produces a result from the last combined-unit, it sends that result to the undivider. It first checks whether it is itself the un-divider.

→ The peers which are not the undivider are now done sending messages in response to the original from the parent.

→ Meanwhile, each time the undivider receives a piece it sends: ReqWorkMsgFromSelf To: worker

- The worker applies the un-divider-unit to the result.

→ In resp, worker sends: RespWorkMsgFromSelf To: scheduler

- This is the same sequence as when one of the combined-units completes, except that the worker checks whether this is the last piece to be undivided (based upon info in the message)

→ In resp, peer sends: InfoComplMsgFromSame To: each other peer (says an undivider done)

→ The peer also sends: InfoComplMsgFromBelow To: the parent (says an undivider done)

→ When the last piece is undivided, the peer sends the same info messages as before, then also sends: RespDataTakeMsgFromBelow, which completes the sequence of steps.

- The parent gets this message, then treats it as if the parent itself had completed applying a combined-unit to the data. Higher-level peers may only apply combined-units which contain a divde-compute-undivide pattern inside them. Leaf-peers, however, may apply any arbitrary combined unit to data, because leaf peers do not sub-divide the data.

6 Results

Results were gathered for a system with four peers. One is a parent, three are in a peer-group below that parent. The parent generates a canned "divide up this data" message (ReqChainDataTakeMsgFromAbove). It picks a random member of the peer-group to be the undivider, and sends all the peers the same message.

The two figures show the output from two of the peers: the parent and the chosen undivider. It shows the initial message going out from the parent, the activity of the messages between the group members, and the final "here's the result" coming back to the parent. (ignore messages 2 and 3 in the peer at 9901, these are debug messages which the author forgot to turn off.)


Sched of Peer at Port: 9901
Message: 1
Got Msg: [DestPort:9901] [Request Chain | data | take | from Level above]
[ID: 1 | whoFrom: 9903]] [Undivider Port: 9901] [data size: 633] [null]
[data owner's port: 9903] [original Msg's ID: 0 | orig who from: 9903]
Message: 2
Handling Msg: [DestPort:9901] [Request Chain | data | take | from Level above]
[ID: 1 | whoFrom: 9903]] [Undivider Port: 9901] [data size: 633] [null]
[data owner's port: 9903] [original Msg's ID: 0 | orig who from: 9903]
Message: 3
To Client Msg: [DestPort:9903] [Request Chain | data | give | from any Level]
[ID: 0 | whoFrom: 9901]] [Undivider Port: 9901] [data size: 211] [null]
[data owner's port: 9903] [original Msg's ID: 0 | orig who from: 9903]
Message: 4
Got Msg: [DestPort:9901] [Response | data | take | from any Level]
[ID: 5 | whoFrom: 9903]] [Undivider Port: 9901] [data size: 211] [null]
[data owner's port: 9903] [original Msg's ID: 0 | orig who from: 9903]
Message: 5
Got Msg: [DestPort:0] [Response | work | take | from own worker]
[ID: 1 | whoFrom: 9901]] [Undivider Port: 9901] [data size: 211] [Main Work Unit]
[data owner's port: 9903] [original Msg's ID: 0 | orig who from: 9903]
main dashed unit was completed
Message: 6
Got Msg: [DestPort:9901] [Info | completion | take | from same Level]
[ID: 2 | whoFrom: 9900]] [Undivider Port: 9901] [data size: 211] [Main Work Unit]
[data owner's port: 9903] [original Msg's ID: 0 | orig who from: 9903]
Message: 7
Got Msg: [DestPort:9901] [Response | data | take | from same Level]
[ID: 5 | whoFrom: 9900]] [Undivider Port: 9901] [data size: 211] [Main Work Unit]
[data owner's port: 9903] [original Msg's ID: 0 | orig who from: 9903]
Message: 8
Got Msg: [DestPort:9901] [Info | completion | take | from same Level]
[ID: 3 | whoFrom: 9902]] [Undivider Port: 9901] [data size: 211] [Main Work Unit]
[data owner's port: 9903] [original Msg's ID: 0 | orig who from: 9903]
Message: 9
Got Msg: [DestPort:9901] [Response | data | take | from same Level]
[ID: 5 | whoFrom: 9902]] [Undivider Port: 9901] [data size: 211] [Main Work Unit]
[data owner's port: 9903] [original Msg's ID: 0 | orig who from: 9903]
Message: 10
Got Msg: [DestPort:0] [Response | work | take | from own worker]
[ID: 5 | whoFrom: 9901]] [Undivider Port: 9901] [data size: 211] [Un divider Unit]
[data owner's port: 9903] [original Msg's ID: 0 | orig who from: 9903]
an undivider unit was completed
Message: 11
Got Msg: [DestPort:0] [Response | work | take | from own worker]
[ID: 6 | whoFrom: 9901]] [Undivider Port: 9901] [data size: 422] [Un divider Unit]
[data owner's port: 9903] [original Msg's ID: 0 | orig who from: 9903]
an undivider unit was completed
Message: 12
Got Msg: [DestPort:0] [Response | work | take | from own worker]
[ID: 7 | whoFrom: 9901]] [Undivider Port: 9901] [data size: 633] [Un divider Unit]
[data owner's port: 9903] [original Msg's ID: 0 | orig who from: 9903]
an undivider unit was completed
done undividing
last chunk, package up data

Figure 1.


Sched of Parent at Port: 9903
Message: 1
Sending to all: Msg: [DestPort:0] [Request Chain | data | take | from Level above]
[ID: 0 | whoFrom: 9903]] [Undivider Port: 9901] [data size: 633] [null]
[data owner's port: 9903] [original Msg's ID: 0 | orig who from: 9903]
Message: 2
Got Msg: [DestPort:9903] [Request Chain | data | give | from any Level]
[ID: 0 | whoFrom: 9902]] [Undivider Port: 9901] [data size: 211] [null]
[data owner's port: 9903] [original Msg's ID: 0 | orig who from: 9903]
Message: 3
Got Msg: [DestPort:9903] [Request Chain | data | give | from any Level]
[ID: 0 | whoFrom: 9900]] [Undivider Port: 9901] [data size: 211] [null]
[data owner's port: 9903] [original Msg's ID: 0 | orig who from: 9903]
Message: 4
Got Msg: [DestPort:9903] [Request Chain | data | give | from any Level]
[ID: 0 | whoFrom: 9901]] [Undivider Port: 9901] [data size: 211] [null]
[data owner's port: 9903] [original Msg's ID: 0 | orig who from: 9903]
Message: 5
Got Msg: [DestPort:9903] [Info | completion | take | from Level below]
[ID: 4 | whoFrom: 9900]] [Undivider Port: 9901] [data size: 211] [Main Work Unit]
[data owner's port: 9903] [original Msg's ID: 0 | orig who from: 9903]
Message: 6
Got Msg: [DestPort:9903] [Info | completion | take | from Level below]
[ID: 4 | whoFrom: 9901]] [Undivider Port: 9901] [data size: 211] [Main Work Unit]
[data owner's port: 9903] [original Msg's ID: 0 | orig who from: 9903]
Message: 7
Got Msg: [DestPort:9903] [Info | completion | take | from Level below]
[ID: 4 | whoFrom: 9902]] [Undivider Port: 9901] [data size: 211] [Main Work Unit]
[data owner's port: 9903] [original Msg's ID: 0 | orig who from: 9903]
Message: 8
Got Msg: [DestPort:9903] [Info | completion | take | from Level below]
[ID: 10 | whoFrom: 9901]] [Undivider Port: 9901] [data size: 211] [Un divider Unit]
[data owner's port: 9903] [original Msg's ID: 0 | orig who from: 9903]
Message: 9
Got Msg: [DestPort:9903] [Info | completion | take | from Level below]
[ID: 13 | whoFrom: 9901]] [Undivider Port: 9901] [data size: 422] [Un divider Unit]
[data owner's port: 9903] [original Msg's ID: 0 | orig who from: 9903]
Message: 10
Got Msg: [DestPort:9903] [Info | completion | take | from Level below]
[ID: 16 | whoFrom: 9901]] [Undivider Port: 9901] [data size: 633] [Un divider Unit]
[data owner's port: 9903] [original Msg's ID: 0 | orig who from: 9903]
Message: 11
Got Msg: [DestPort:9903] [Response | data | take | from Level below]
[ID: 17 | whoFrom: 9901]] [Undivider Port: 9901] [data size: 633] [Un divider Unit]
[data owner's port: 9903] [original Msg's ID: 0 | orig who from: 9903]
Handling response from undivider
Which means the test succeeded.

Figure 2.

7 Improvements

7.1 Leaving pieces of larger chunks distributed across processing nodes

One improvement is to allow the data-chunks to remain divided, when convenient. In this scheme, the data-chunks passed among higher-level peers would consist of lists of smaller chunks. The division of these chunks would involve dividing up the lists. When such a chunk reaches a leaf-peer, that peer requests one or a few of the elements of the list from peers holding the actual data. (note from author: this blank space appears to be a bug in my version of TexMacs)

This complicates the division and undivision process. In some cases, when nested loops exist in the undivision-unit, for example, it may be advantageous to collect the pieces onto the undivider peer and construct a single result-pieces. In this way, multiple sizes of data pieces may be included in a single chunk.

Peers asking for the wrong piece must still be detected. This may be accomplished, for example, by requiring that any division of a given list of data-pieces must have at least one higher-level peer which all requests for the data-pieces pass through. The division algorithm would then decide and specify which higher-level peer all piece-requests must pass through (in deterministic replicatable way).

This higher-level peer then gets the original list of pieces and tracks each request, to ensure that all peers in a group came to the same conclusion about how to divide up the data.

Allowing the data-pieces, which make up a larger chunk, to stay on the leaves in this way may improve performance. The performance gains would come from reducing redundant transfers of data. If the larger chunks are physically constructed, then the pieces must be sent to the undivider machine. Now, if this chunk is used in another input-set, it has to be divided up again. Each leaf-level peer getting a piece will then request it from the undivider. Thus the actual data moves twice, once to the undivider, and once again to the new leaf-level peer. On the other hand, with undividers which don't require the contents of the data, only meta-information about it, only a single transfer of the actual data takes place, at the time the new leaf-level peer requests it.

7.2 Data retention for reliability, correctness, and failure detection

To prevent the loss of partial results due to the failure of a processing node, some scheme may be used which retains "ancestors" of results until suitable downstream results are completed on distinct processing-nodes. The idea is to have at least two nodes with a "recoverable" form of the data at all times. Thus, a processing-node keeps data until it is confirmed that two other processing nodes now have some form that the most recent results, on the third node, can be reproduced from the results on the second node.

The communication involved in this protocol would detect lost messages eventually, and allow recovering from machines going down.

Recovery is possible if an invariant is maintained. The invariant says that if a descendant, call it A, shares a processing-node with any ancestors of a second descendant, call it B, then A does not contribute towards the reproducibility of B. Only when a set can be formed of descendants which have the reprodicibility property for all lower-descedants of a given ancestor, can that ancestor be safely deleted. This invariant is easier to calculate than might be guessed, because the flow-graph is fixed. The back-end compiler may calculate a static function which takes in a list of descendant result-pieces and processing-nodes they reside on and return a yes or no answer for a given ancestor (possibly.. needs more analysis).

A simple way of saying this is that its a sort of worm-hole routing, where the tail of the worm dissapears as the head makes progress. The complication is that the heads of many worms combine together. To show the correctness of this method, it must be shown that the combined-head may be reproduced from the middle-links. As long as enough middle links are on different processing-nodes from the combined-head, the combined head may be lost. The middle-links are then used to reproduce it on another processing node.

7.3 Delaying actions to increase their prediction accuracy

The utility of each action can be optimized by estimating prediction accuracy and modelling the cost of misprediction. Tracking the statistics of flight times between peers can be used to estimate prediction

accuracy. A given peer can then simply acts as though a particular message took longer to get there before taking action implied by that message. In the meantime, any update messages with previous time-stamps have a chance to arrive. The amount of time to wait between receiving a message and taking action is determined by optimizing the utility of waiting.

The chance of misprediction decreases the longer a peer waits, and can be estimated using the flight-time statistics.

The cost of misprediction also changes with time, and depends upon the type of message and the called-for action. The cost of mispredicting a data-request action decreases with time because less network capacity is used when the transfer time decreases, assuming the data transfer is still in progress when the discrepancy is detected, also the collision rate decreases when fewer non-useful packets flow, which increases the effective bandwidth for the correct transfers. The cost of mispredicting a computation action, however, is zero if it does not interfere with other, higher utility computation actions.

Making the computation queue in the worker a priority queue, which chooses the highest utility computation next implements this. Older messages naturally gain higher utility due to their improved prediction rate, which prevents starvation.

7.4 Redundancy of higher-level peers

For the sake of reliability, possibly performance and somewhat security, each higher-level peer may be implemented with redundancy. In this case, it would be duplicated on several processing nodes, and the duplicates participate in a voting scheme. The degree of replication would increase as the level in the hierarchy increased, with root being the most widely replicated.

The increased reliability is self-evident.

Performance improvement would acrue in a large system, with hundreds of thousands of nodes, from selecting a quorum of processing nodes close, in a network latency measure, to each other and to the data under consideration. The replicated peer would also have higher throughput, due to multiple quorums operating simultaneously on different processing-nodes.

The increased security would come from the ability to detect the processing nodes whose peers consistenly give wrong answers. For example, this would catch virally infected nodes which were not involved in a coordinated attack. The ability to detect misbehaving processing nodes may also serve as an element in a more sophisticated security scheme.

8 Conclusion

We have seen how the central idea of CodeTime, the addition of coordination-constraints to dataflow, has resulted in the useful properties: self-contained chunks of work, and asynchronous computation. We have also seen how each property has been used to provide benefits.

Self-contained chunks of work allowed the divide-compute-undivide pattern that lets the granularity of computation be changed dynamically. This benefits software economically via a one-size-runs-efficiently-on-all distribution of code. Old code runs well on new machines, and bigger machines.

The asynchronous computation model lets the run-time system produce correct results without any synchronization operations (footnote: time-stamps and the resolution process takes the place of syncs). We have seen how this makes the speculative method shown here attractive. The speculative method has allowed scaling to very large systems, due to the small number of control messages, and it allows optimizing the computation by choosing the highest utility computation at each moment, which we have shown is guaranteed to outperform, on average, a scheme which waits for certainty (an ack or a lock). The asynchronous model also benefits distributed systems and high-end computers in which barrier operations are expensive.