An Extensible Virtual Machine Design for the Execution of High-level Languages on Tagged-token Dataflow Machines

Mathijs Saey Joeri De Koster Jennifer B. Sartor Wolfgang De Meuter
Software Languages Lab, Vrije Universiteit Brussel, Belgium
{mathsaey,jdekoste,jsartor,wdmeuter}@vub.ac.be

Abstract

Dataflow models are a promising platform for programming language implementation due to their ability to extract the latent parallelism present in a given program. In spite of this, no modern-day language has emerged which leverages this property of implicit parallelism. As a first step towards the creation of such a language, we introduce a virtual machine design which is based on the tagged-token dataflow model. Notably, this design offers the fundamental concepts of the underlying model as first-class entities, which makes it possible to support various high-level language features without the need for any additional runtime support.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Run-time environments

Keywords Dataflow, Extensible Virtual Machine design

1. Introduction

The dataflow model (Johnston et al. 2004) is an attractive model for the expression and execution of highly parallel programs due to its lack of global memory and its data-driven style of execution. In spite of these attractive properties, there are no modern high-level programming languages currently in use which leverage this model and its implicitly parallel style of execution.

We investigate the creation of a virtual machine (VM) which can serve as an execution platform for these implicitly parallel languages. Our VM uses the *tagged-token* dataflow (Arvind and Nikhil 1990) model of execution in order to support concurrent code reentry and recursion. This model uses *tags* to differentiate between tokens with a different execution context. Our VM can leverage these tags to enable the implementation of high-level language features such as exceptions and closures without the need for any additional runtime components.

In this paper, we provide a brief introduction to the dataflow model (Section 2) and the tagged-token style of execution (Section 3). Afterwards, we use these concepts to present the design of our VM (Section 4).

MoreVMs'17 April 3, 2017, Brussels, Belgium

2. The Dataflow Model

In the dataflow model, every program can be represented as a graph. Every operation in the program is represented by a node, while the edges between the nodes represent *data dependencies* between these operations. For instance, the following function, which calculates the distance between two coordinates, is represented by the graph shown in Figure 1.

```
(define (distance x1 y1 x2 y2)
  (sqrt (+ (square (- x2 x1)) (square (- y2 y1)))))
```

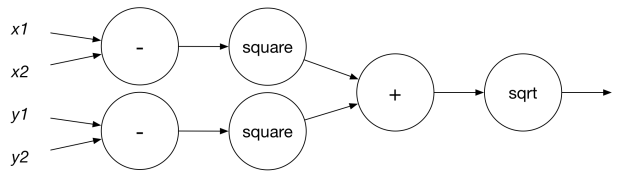

Figure 1. Dataflow representation of the *distance* function.

While evaluating this function, an operation can be executed when all of its input data (represented as a *token*) is available. When an operation is executed, it *consumes* all of its input tokens, and potentially produces a new token which contains the value which results from applying the operation to its input tokens. This data is sent to the next node in the program graph. For example, when calling the distance function with four inputs, both of the '`-`' nodes could be executed. Executing any of these nodes would result in a new token, which would get sent to the '`square`' node connected to this particular '`-`' node. Note that the dataflow model allows both of these '`-`' (or '`square`') operations to be executed in parallel: the dataflow model only imposes execution order based on data dependencies present in a program.

This model does not cover what happens when code is *reentrant*, i.e. when multiple instances of the same part of the program are executed concurrently (e.g. when two invocations of the same function are active at the same time). To solve this, multiple practical versions of the dataflow model have been created (Veen 1986). We focus exclusively on the *tagged-token* dataflow model (Arvind and Nikhil 1990).

3. Tagged-token Dataflow

The tagged-token dataflow model tackles the reentrancy problem by using *tags* to differentiate between tokens from different execution contexts. Tags are added to every token and contain an *address*, a *port* and a *context*. Program data encapsulated by a token is called a *datum*. The address identifies the destination (i.e. operation) of a token, while the port determines which input of an operation a token goes to. Finally, the context uniquely identifies the execution context (e.g. a particular function invocation) of a token.

Traditionally, a tagged-token processing pipeline consists of a *token queue*, which contains the tokens that are ready to be processed, a *matching memory*, which stores the partial input of each operation for each context, an *execution unit*, which executes operations by applying the operations to the data of its input tokens, and a *tokenizer* which wraps the results of these operations in a tagged token.

In these pipelines, features such as function calls are typically implemented by a dedicated set of special operations, which may have access to their own memory. The issue with this design is that language features which are not explicitly supported by the VM can only be built by expressing them in terms of existing features, at the cost of using extra operations, or by adding new 'special' operations to the VM, which is a tedious and difficult process.

4. Tagged-token Operation Design

To address this issue, our VM explicitly offers the core components of the tagged-token dataflow model as first-class entities which can be manipulated during the execution of operations. This is done in such a way that it is impossible to access data from different contexts, which ensures that the semantics of the tagged-token dataflow model are not violated. In turn, this guarantees that the parallel properties of the dataflow model can still be exploited.

The design of our runtime is similar to a traditional tagged-token runtime engine, with the key difference that operations do not operate on token data. Instead, they operate on their set of tagged input tokens, a part of the matching memory, a context manager and a context-specific key-value store. The following paragraphs describe these entities and how they can be leveraged to build arbitrary new operations.

First and foremost, operations get to read all of the data encapsulated in input tokens and their tags. They also have the ability to create tokens with arbitrary addresses, ports and contexts. Operations can use this feature to dynamically determine the destination and execution context of any token, which can be used to create conditional statements or to pass arguments to a function invocation. Furthermore, a datum may consist of arbitrary addresses, ports, or contexts, enabling operations to modify tags based on the value of a datum (i.e. program data). Since operations are directly responsible for the creation of tagged-tokens, the tokenizer, which is present in traditional tagged-token architectures, is no longer required in our VM. Instead, tokens produced by operations are automatically added to the token queue.

Second, operations get read-only access to the current state of the matching memory for the context with which it is being executed. This feature makes it possible to dynamically capture the environment of an execution context (e.g. to implement features such as closures).

Furthermore, operations can spawn new execution contexts, enabling operations to make any piece of a program reentrant, which is required to implement fundamental language features such as (recursive) function calls and loops.

Finally, operations get access to a key-value store shared with any operation that is executed with the same context. This store generalizes the memory which 'special' operations need to function in a traditional tagged-token dataflow VM. Operations can use this memory to store execution information such as the return address of a function call.

Combining the ability of operations to arbitrarily read and produce tagged-tokens with the set of runtime components we presented above enables the expression of various high-level language features in terms of tagged-token operations, without the need to modify the internals of the VM. For instance, if one wishes to add support for calling functions to our VM, it suffices to spawn a new context, send the data to the first operation of this function with this new context, and to store the return address and original context in the context-specific key-value store. Returning from this function can be done by retrieving the return address and original context from the store, after which the result token can be sent to the return address with the original context. Other features, such as exception handling, higher-order function calls or closures can be implemented in similar ways.

5. Conclusion

We have shown how our VM leverages the building blocks of the tagged-token dataflow model to allow language designers to express various constructs in terms of operations. This removes the need to modify the internals of the VM. This allows us to build implicitly parallel languages on top of a flexible dataflow virtual machine.

Acknowledgments

This work is funded by the *Flamenco* project of the Flemish agency for Innovation by Science and Technology.

References

K. Arvind and R. S. Nikhil. Executing a Program on the MIT Tagged-Token Dataflow Architecture. *IEEE Transactions on Computers*, 39(3):300–318, 1990.

W. M. Johnston, J. R. P. Hanna, and R. J. Millar. Advances in Dataflow Programming Languages. *ACM Computing Surveys (CSUR)*, 36(1):1–34, 2004.

A. H. Veen. Dataflow Machine Architecture. *ACM Computing Surveys (CSUR)*, 18(4):365–396, 1986.