A Proposal for Targeting Streaming Languages with the Value State Dependence Graph

James Stanier
Department of Informatics,
University of Sussex
j.stanier@sussex.ac.uk

Des Watson
Department of Informatics,
University of Sussex
desw@sussex.ac.uk

Abstract

The Value State Dependence Graph is a sparse data-dependence IR that has been investigated as a midpoint in compilers producing sequential code for uniprocessors. However, we argue that parallelism is an important feature that has been largely overlooked. We conjecture that the VSDG is exploitable for parallelism through partitioning, and investigate the feasibility of generating streaming languages via a VSDG implementation, possibly bridging the gap between streaming languages and von Neumann-style languages such as C.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Compilers

General Terms Algorithms, Languages

Keywords VSDG, StreamIt, stream programming, intermediate representations

Introduction

The Value State Dependence Graph (VSDG) is an intermediate representation for modeling programs using data dependencies. It can be seen as a spiritual successor to the Value Dependence Graph[4], with the addition of "state" edges to allow only the essential sequential dependences in a program to constrain the ordering of instructions. The IR was introduced by Johnson[1] in 2004, with further developments by Upton[3] and Lawrence[2] in 2006 and 2007 respectively.

The aforementioned work focuses on the difficulties of producing sequential code from the VSDG. Sequentializing the graph in an optimal fashion was proved to be NP-complete[5], resulting in a new theoretical compiler architecture being proposed by Lawrence in order to overcome this problem. However, we feel that work to date has overlooked the fact that the structure of the VSDG may be exploitable for parallelism. We shall now briefly introduce the graph, and then consider how partitioning could proceed.

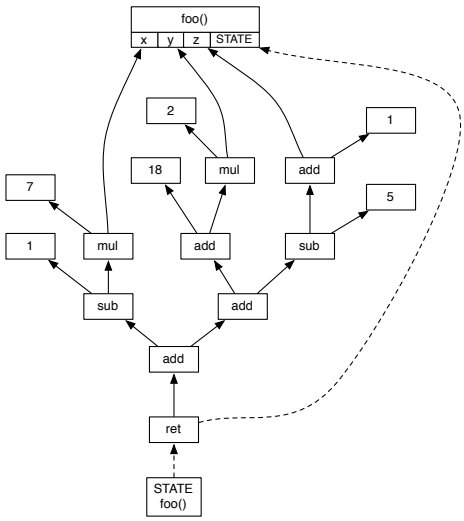
Partitioning the VSDG

In this section, we explore the feasibility of partitioning the graph into instructions that could be executed in parallel. Firstly, we present an example program in source code and as a VSDG. We then define properties of the VSDG that afford parallelization. Then, we sketch an algorithm for doing the partitioning.

The graph diagram in Figure 1 illustrates a majority of the main VSDG components. Solid lines in the graph show data dependencies and dashed lines show state dependencies. Data dependency edges illustrate the values that must be consumed to evaluate the instruction they come from. State dependencies encode the essential sequential order of execution; the minimal control information needed to represent the program semantics.

```
int foo(int x, int y, int z) {
   x = x * 7;
   a = x - 1;
   y = y * 2;
   b = y + 18;
   z = z + 1;
   c = z - 5;
   return a + b + c;
}
```

(a) A simple function.



(b) The corresponding VSDG.

Figure 1. An example function.

Since the VSDG represents programs in such an adaptable manner, there exist properties that can be exploited for parallelism.

PLDI '09 June 15–21, 2005, Trinity College, Dublin.

Many expressions in the VSDG representation have no side-effects. Any part of the graph that does not read or write to the state can be considered *referentially transparent*. It follows that any part of the graph that has this property can be evaluated and replaced with a result without affecting the trace semantics of the program.

Parts of the graph that do interact with the state of the program must wait for other sections of the program to evaluate in order to be evaluated themselves. Therefore we see state-interacting nodes as *barriers*, where the subgraph contained between two barriers is a *step*. The intuition behind this name is that one state is live at a time, and the evaluation of one state followed by the next represents the main "steps" in the computation. Within each step, there may exist some parallelism. In Figure 1 a step is trivially the whole function, since the content is pure – it is contained within the enclosing region of the state edge. Within each step, we can apply an algorithm for discovering chunks of the computation that can be parallelized. A useful property of non side-effecting statements is that they are always structured as a tree in the VSDG. We perform a reverse breadth-first search from the current live state, marking each instruction found with the current level of the search. For each level of the search, there will exist some number of instructions that reside on that level. Due to the tree structure, we reason that these can be executed in parallel (Figure 2).

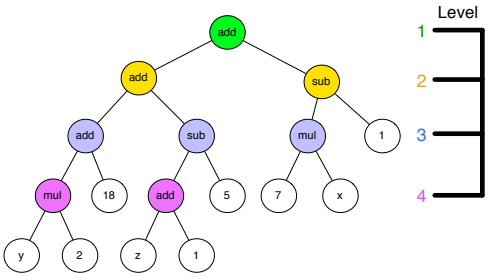


Figure 2. Non side-effecting expressions from Figure 1 form a tree. All expressions on a given level can be executed in parallel.

From the VSDG to streaming languages

Streaming languages such as StreamIt[6] are a development towards representing programs as explicit data channels of communication. As well as being an ideal way of representing streaming applications, they serve as a step towards programming effectively for multicore machines[7]. In StreamIt, a program is represented as independent filters which communicate via explicit data channels.

We now consider a way of targeting streaming languages with our partitioned VSDG. StreamIt provides three single-input, single-output structures in addition to the filter: a *pipeline*, which is a string of filters; a *splitjoin* which splits parallel computation and a *feedback loop* where the output feeds back into the input. The whole-program VSDG forms a StreamIt pipeline, where the order of execution is that which is enforced by the state edges in the graph. Each step in the program becomes a filter in the pipeline. This ensures that the I/O semantics of the program are maintained. If it is the case that a step contains non side-effecting parallel partitions, then the filter in the pipeline for that step becomes that of a splitjoin structure to execute these partitions in parallel, in a level-by-level manner as marked by the search algorithm, as shown in Figure 3.

Bridging the gap

Languages such as C that target uniprocessor machines have been in use for a very long time. If it is the case that the streaming languages paradigm becomes a mainstream standard, then it follows that a vast quantity of existing programs will not be able to take advantage of the benefits that streaming languages bring. Constructing the VSDG straight from C has already been accomplished[1]. If it is the case that targeting streaming languages with the VSDG is a feasible option, then it presents a very interesting opportunity for extracting parallelism from existing non-parallel programs.

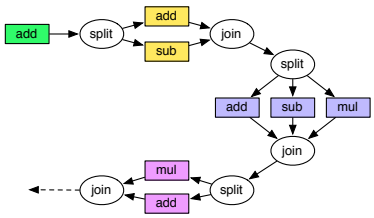


Figure 3. Instructions arranged as StreamIt filters.

Ongoing work

There are two VSDG constructs omitted from this paper. The first is the γ-node which acts as a conditional branch. Here, a condition C is evaluated, and then data is consumed through the T or F edge depending on the result of the condition. Optimizations such as if-conversion would make these suitable for this style of partitioning, or branches could even be executed in parallel. The other construct is loops. Lawrence proposed that loops should be infinitely unrolled structures in the VSDG. Since these infinitely unrolled structures are also tree shaped, we conjecture that the partitioning would be applicable here too. Work is being undertaken using the LLVM compiler to explore a variety of compilation ideas using the VSDG. Before adopting the approach this paper takes to arranging StreamIt filters, more research must be done to ensure that the overhead of the splitting and joining operations is less than the overall speedup through parallelism. For example, it will be worth considering larger regions of multiple instructions for partitioning, using techniques already presented in the literature.

Acknowledgments

We would like to thank Philip Brisk, Christopher Gautier and Paul Biggar for their comments and helpful discussion.

References

[1] Neil Johnson. Code size optimization for embedded processors. Ph.D. thesis, University of Cambridge, 2004.

[2] Alan Lawrence. Optimizing compilation with the Value State Dependence Graph. Ph.D. thesis, University of Cambridge, 2007.

[3] Eben Upton. Compiling with Data Dependence Graphs. Ph.D. thesis, University of Cambridge, 2006.

[4] Daniel Weise, Roger F. Crew, Michael Ernst and Bjarne Steensgaard. Value dependence graphs: representation without taxation. In *Proceedings of POPL '94*, ACM, 1994.

[5] Eben Upton. Optimal Sequentialization of Gated Data Dependence Graphs is NP-Complete. In *PDPTA '03*, 2003.

[6] William Thies. Language and Compiler Support for Stream Programs. Ph.D. thesis, MIT, 2009.

[7] Michael Gordon, William Thies and Saman Amarasinghe. Exploiting Coarse-Grained Task, Data, and Pipeline Parallelism in Stream Programs. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, ACM, October 2006.