

# Using vfAnalyst to Understand Program Behavior

The continued unabated need for increased computing power by means other than increased clock speeds has forced a move to multicore processors. Instead of running a single processor twice as fast, two cores can theoretically provide twice the work. If two completely independent tasks are being performed, then this equation can be relied on. But if a single large task is to be decomposed in a way that allows many independent processors to do the work, it is very unlikely that the many cores will provide the same performance as one core running that many times faster.

The benefit of taking a single sequential program and breaking it apart to run various pieces in parallel depends on whether these pieces rely on each other for completion. This is driven by dependencies within the program, and understanding such dependencies is critical to creating an effective parallel implementation that is functionally identical to the original sequential version. This whitepaper will describe the relationships and dependencies that exist, and will highlight them in the context of a new analysis tool called vfAnalyst that can be used to understand the behavior of a program.

## Introduction

A typical sequential program will consist of elements that can be run independently of each other and elements that must be run in order. Amdahl's Law states, more or less, that the amount of performance improvement that can be gained through parallel computation is effectively limited by those portions of the program that have to be run in order. For  $n$  processors, the only way to have the program run  $n$  times as fast is if it can be broken up into  $n$  mutually-independent chunks of equal length; this is almost never possible. That means that programs will have to be satisfied with something less than a speedup of  $n$  times. The design challenge becomes one of trying to maximize that speedup.

In a given program, there are generally things whose performance is critical and things that are less critical. Getting the critical portions to run as quickly as possible involves ensuring that the non-critical parts are out of the way. All efforts at parallel computing rely on removing those dependencies that can be eliminated and working around the ones that remain. Understanding where all of those dependencies lie is nontrivial, and missing dependencies can result in a parallel version of a program that is functionally different from the sequential original. This has created the need for a tool that can identify and present the behaviors and dependencies of a program so that a correct, effective parallel implementation can be created.

The vfAnalyst tool is intended to provide software writers with a clear, easy-to-interpret picture of how a program behaves, where the performance bottlenecks are, and, uniquely, which parts of the program depend on each other. This information can be used to reduce the number of dependencies in the program and also to create an effective parallel

implementation of the program. The scope of this paper is on understanding the characteristics highlighted by the vfAnalyst tool; the use of that information for creating parallel implementations will be covered in a separate paper.

The programming language used to write the program in theory does not affect the analysis of the program, although, in theory, a perfect parallel language (which doesn't exist) wouldn't need an analysis tool; the dependencies would be obvious by inspection. ANSI C is very far from that ideal, and is the most commonly used language that benefits enormously from analysis, so this discussion will be placed in that context. But the concepts hold regardless of language, and vfAnalyst could be enhanced to handle other languages as well.

## Displaying the properties of a program

Program behavior can be complex, and it can therefore be useful for that behavior to be depicted in a straightforward manner. While this whitepaper is intended to concentrate on the concepts underlying the graphic depictions, examples are also provided to correlate the concepts to their respective depictions in the vfAnalyst tool. In order for those examples to make sense, it is helpful to understand the high-level structure of the vfAnalyst display.

The vfAnalyst display comprises several basic elements as indicated in Figure 1. Two of the most important are the *1-D profile* (or simply the *profile*) and the *2-D profile*. In addition, shown below are the *dependency pane* and a *property pane*. All of the concepts described below will be depicted in one (or more) of these elements.

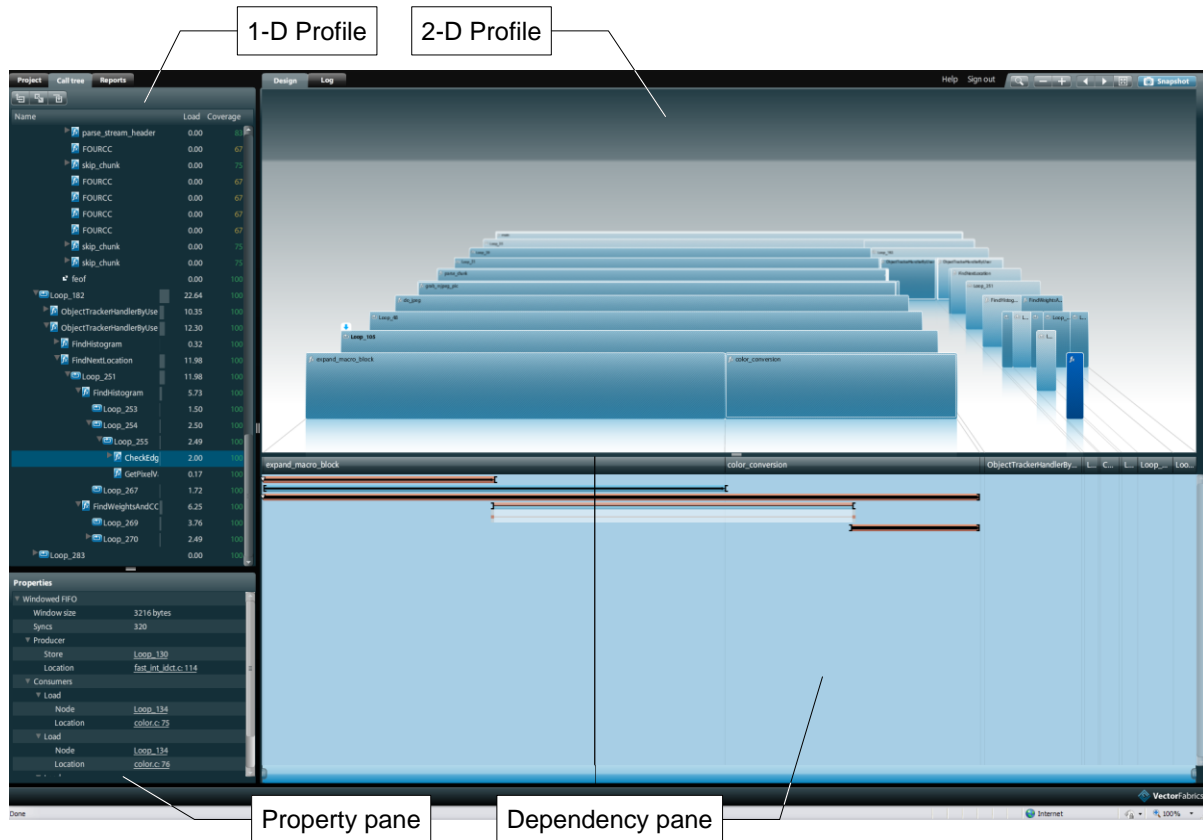


Figure 1. The vfAnalyst display

### Some terminology

There are some closely-related terms that can cause some confusion. It's best to clear some of those up first.

#### Multi-what?

The disciplines associated with parallel programming can span such terms as “parallel computing,” “massively parallel computing,” “grid computing,” “multiprocessor,” and “multicore,” to name a few. All of these have specific implications: “multicore” tends to mean multiple computing elements within a single chip; two chips, each with a single processor, has been called “multiprocessor.” “Grid computing” tends to involve multiple computer boxes interconnected by a network. But not everyone one uses the terms in this way.

In general, these differences involve communication between actual computing cores and memory coherency. These are distinctions that are not important for the general topic of creating parallel versions of sequential programs; the techniques described here can apply to any of these parallel processing contexts. However, the actual timing of execution can vary widely depending on how the processors communicate and how their memories relate, so this can't be ignored completely when deciding on an implementation.

#### Tasks

There are two fundamental kinds of component that can operate in parallel with each other: *threads* and *processes*. Threads are subordinate to a process and share a common memory space; processes are completely independent. A multicore program can be implemented as a single process with multiple threads, as a group of small single-thread processes in an asymmetric multiprocessing (AMP) environment, using inter-process communication (IPC) to share any necessary information, or as a combination of both.

In this document and in the vfAnalyst display, the word *task* will be used to indicate a section of a program that is to be executed concurrently with other sections – that is, a thread or process. The details involved in the distinction between a thread and a process will not matter for the bulk of the discussion.

With the first release of vfAnalyst, only a single task will be supported. On a subsequent release, multiple tasks can be analyzed.

### What level of parallelism?

When creating a parallel version of a program, the first obvious question is, at what level should opportunities for parallel execution be investigated? At the lowest level, instruction-level parallelism can be exploited to reorder and parallelize low-level operations. However, this depends greatly on the architecture of the target processor and is something that compilers already focus on; it's hard to add additional benefit. At the other extreme, if very large chunks of a program are examined, they will almost always be mutually dependent if the computing involves anything nontrivial.

A useful mid-point to use is the processor-memory boundary. While the processor itself and its registers are managed by the compiler, memory architecture is not explicitly considered by the compiler; this becomes a level where investigating relationships is useful.

The two critical operations here are saving to memory and reading from memory: *store* and *load* operations. Variables kept in processor registers will be managed by the compiler, but as soon as they are stored into or loaded from memory, we have an opportunity to manage the dependencies that arise from these operations.

This concept can be extended further to include peripherals (which are often memory-mapped): any time a variable leaves the scope of the processor (and compiler), it becomes an item of interest in our analysis. The use of stores and loads as well as other peripherals will be explored in detail below.

The other question that arises is how large a chunk of code can be analyzed. It is very common to study the behavior of programs at the function level. But much of the hard computing work, especially in embedded systems, is accomplished through the use of loops. These are of critical interest when determining how to parallelize a program, so vfAnalyst identifies not only *functions*, but also *loops*. vfAnalyst also recognizes *intrinsic*s, which are discussed below.

The 1-D profile section of the vfAnalyst display provides a hierarchy that represents the structure of a program, and the lowest-level units that can be represented there are functions, loops, and intrinsic. Because these elements share common traits within vfAnalyst, the name *invocation* will be used here to refer to any of them.

This provides an execution hierarchy: the task, which will run as independently as possible of other tasks, and is always the top level of the hierarchy; the functions within a task, being the second level; and any functions, loops or intrinsic that may exist within each other, making up further levels of hierarchy. vfAnalyst displays each of these elements in a distinct way, as will be described below.

### Static and Dynamic analysis

Much of the behavior of a program can be learned simply by analyzing the code. This is referred to as *static analysis*. However, this can miss critical behaviors, especially regarding how pointers and other real-time memory-related operations behave. Understanding these behaviors requires *dynamic analysis* – that is, watching what happens while a program executes on a specific set of data. vfAnalyst performs both static and dynamic analysis using test data that you provide.

#### Test coverage

During dynamic analysis, vfAnalyst logs the behavior of the program, noting and recognizing various activity patterns, and deriving from those various dependencies and relationships. For this reason, it is important that as complete a dataset as possible be used as input to the program so that all possible code branches are taken and all possibilities are covered. To help you understand how complete your dataset is, each invocation is given a test coverage score

that indicates how many of the lines of code are tested by the data being executed. Invocations with low coverage are highlighted.

It is important to remember that *the analysis that Vector Fabrics' tools provide will only be as accurate as your test set allows*. The tools will rely on you to ensure that they're getting a full picture of operation. Using a simplistic example, if your program behaves differently depending on whether a key value is odd or even, and if the test set only happens to exercise the odd numbers, then the resulting analysis will only reflect the effects of odd numbers. If you see analysis results like this that seem counter-intuitive, you should review your test coverage to ensure that the tools really have a complete picture.

The flip-side of this is that, if you are reassured that your test set is good, the tools can identify real patterns and characteristics that you never realized existed, allowing you to exploit them in a fashion that would have been impossible without the tools.

### Compute load and delay

The concept of "time" is challenging to represent generically. Wall clock time elapsed during execution depends entirely on the system doing the computing. But the job of parallelizing the program needs to be done in a manner that doesn't specifically rely on a processor clock speed. Therefore the vfAnalyst concept of "time" is relative.

There are two ways of looking at this. One is the *compute load*, which represents the total amount of work done by a particular function or other entity. The other way is to specify the delay of a part of the program relative to the whole program. In vfAnalyst this is referred to simply as the *delay*, but it is actually the percentage delay, where *main()* has, by definition, a delay of 100.

While compute load and delay are related, there is a critical distinction. Let's say that one function *alpha()* calls two other functions *beta()* and *gamma()*. *Beta()* and *gamma()* each do some amount of work, and the amount of work that *alpha()* does is the sum of the work done by *beta()* and *gamma()* (plus whatever other code is inside *alpha()*). The amount of work that *alpha()* does is the same whether *beta()* and *gamma()* run one after the other or run in parallel.

The delay, on the other hand, is not the same. If *gamma()* runs after *beta()*, then *alpha()* will take longer to execute than if *beta()* and *gamma()* can be made to run at the same time. The compute load is invariant, but the delay is impacted.

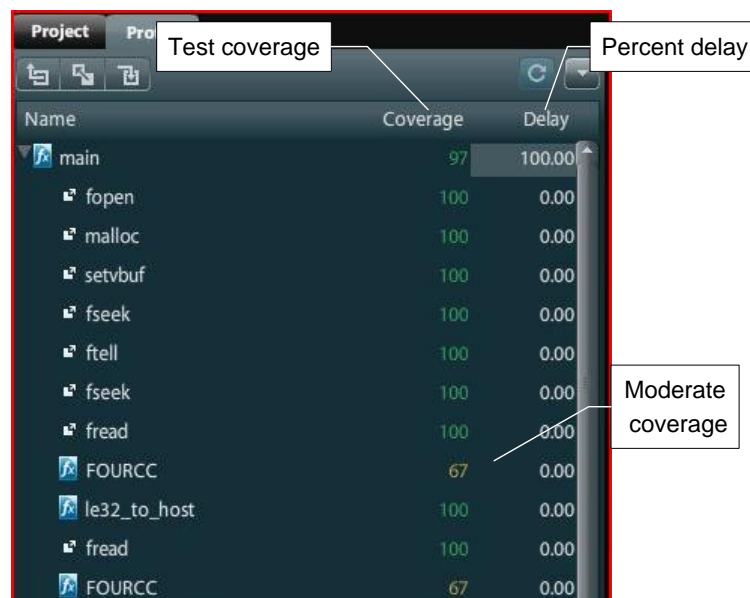
The vfAnalyst display shows the amount of work done by a function as delay (i.e., percent delay) so that the effects of parallelization can be seen. The actual amount of time it takes to complete this work will depend on other variables like processor clock speed and memory access speed..

It's also important to bear in mind that vfAnalyst is system-agnostic: it has no awareness of the computing platform, and so the delays are estimates. It focuses only on the intrinsic concurrency of the program based on its structure. This is the critical first step in

understanding how a program can be parallelized as well as for identifying any structural elements of the program that get in the way of parallelization. vfSoftware (and some other future tools) will add system-awareness to include those effects in a more accurate determination of delay.

It is possible to confuse the compute load with a measure of the relative number of lines of code in a portion of the program. This is not the case; the compute load reflects actual execution.

Figure 2 shows how vfAnalyst depicts the hierarchy of tasks and invocations in the 1-D profile, along with their delays and coverage. The delay is accompanied by an appropriately-sized gray bar for visual reinforcement. You can filter out extremely low-load invocations to remove clutter and focus only on high-load invocations. Test coverage numbers that are low are indicated by different number colors.



**Figure 2. Compute load and test coverage in the 1-D profile**

Because the 1-D profile shows actual execution (not file or class structure), a given function may occur in more than one place if it is called in different parts of the program. The delay in each such instance will reflect the delay in that context only; the delay of each contributor to a higher level of hierarchy will sum to that higher level (with some possible round-off error). However, the test coverage for each instance will reflect the cumulative coverage for all instances, and will show the same coverage number for each instance, as shown above, even though each instance may individually contribute a different amount to the overall coverage.

The information for a given invocation, is also available in a separate properties pane, as shown in Figure 3.

| Properties              |                         |
|-------------------------|-------------------------|
| ▼ Loop_649              |                         |
| Loop                    | Loop_649 (IDCT)         |
| Compute load            | 20.2 %                  |
| Line coverage           | 100.0 %                 |
| Iteration count         | 8                       |
| ▶ Iteration time        |                         |
| Source location         | fast_int_idct.c:102-114 |
| ▶ Induction Expressions |                         |

**Figure 3. Loop properties**

The properties pane includes additional information about the invocation, which, in this case, is a loop. Some of the information is repeated from the 1-D profile, but you have additional information about the iteration count (which is an average of all encounters, and so might be fractional), induction expression, and location of the loop in the source code.

The iteration count as shown is the average number of times the loop actually iterates overall. In the case shown here, it's not an integer, which indicates that the loop is called as part of a branching structure of some sort (discussed below), and that the loop does not execute every time. This can help identify loops that, while perhaps having a high loop count, execute infrequently, and therefore have less impact on performance than would be indicated by the loop limits in the code.

The source locations reflect the file and line numbers, but are actually clickable to take you to the portion of the source code where the loop is defined.

Induction expressions are discussed below in the context of loop-carry dependencies.

### Invocations and branches

This section describes how the various invocations are represented within vfAnalyst, along with some other constructs that vfAnalyst can identify.

#### Functions

Functions are indicated as shown in Figure 4. The horizontal width of the function indicates the total delay of the function, which includes the delays of all functions called by the function, all iterations of all loops within the function, and any recursion within the function. For this and most of the elements being illustrated below, the icon in the upper left corner of the rectangle is key to identifying what is being represented. For functions, the  $fx$  is the indicative icon.

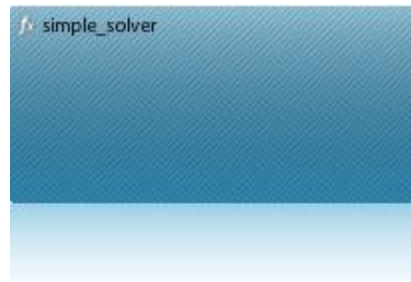


Figure 4. A function

Recursion is indicated as shown in Figure 5. The white bounding box indicates the delay contributed by all instances of recursion.



Figure 5. A function with recursion

### Loops

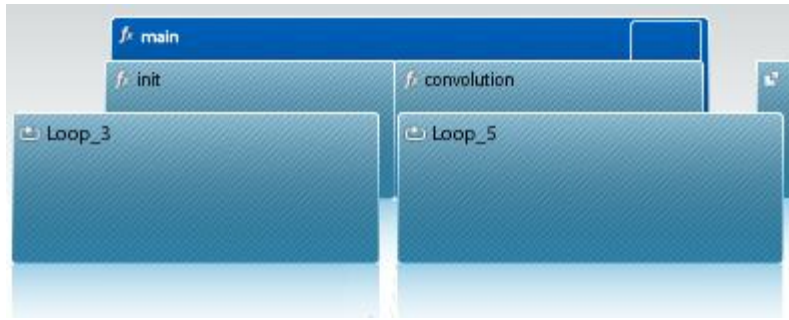
Loops are indicated as shown in Figure 6, with a loop graphic being the identifying icon. The horizontal width of the loop indicates the accumulated delay of all iterations of the loop. However, the positions of various events within the loop are placed on the box in proportion to the event in a single iteration. Because loops aren't named in C programs, vfAnalyst gives each loop a name; in the figure, the loop has been named *Loop\_203*.



Figure 6. A loop

### Nesting

Because loops can contain functions and more loops, and because functions can also contain loops and more functions, vfAnalyst uses a 3D effect to indicate these nesting relationships. For example, Figure 7 shows a function *main()* that contains two sub-functions *init()* and *convolution()*. Function *init()* contains loop *Loop\_3* and function *convolution()* contains loop *Loop\_5*.



**Figure 7. Function and loop nesting**

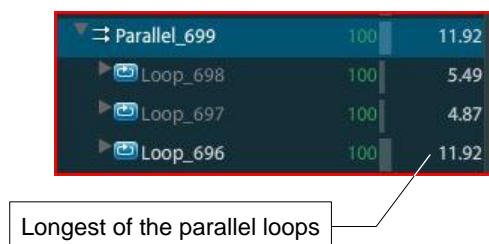
The 2-D profile provides an easy way to see the nesting of elements of the program and their relative contributions to compute load. If a more precise horizontal alignment of events is needed, however, a special *linear view* can be used to project all of the elements onto one dimension, as shown in Figure 8. This shows the proportional timing as *main()* starts, and then sub-function *init()* starts, indicated by the small gap and followed immediately by loop *Loop\_3* starting up. After *Loop\_3*, there is a small gap where *init()* finishes and another where function *convolution()* commences. *Loop\_5* then carries on as *Loop\_3* did, etc.



**Figure 8. Linear view for aligning events in time**

## Parallel invocations

If multiple invocations have been parallelized, they will be displayed as being “contained” within a *parallel* invocation. This structure will be shown in the 1D profile as part of the hierarchy, with the parallelized invocations being “children” of the parallel container. This is shown in Figure 9, with the longest of the parallel loops (*loop\_696*) highlighted.



**Figure 9. Parallel invocation in the 1D profile**

The parallel structure will also be shown in the 2D profile, but in order to avoid clutter, only the longest of the contained parallel invocations is shown by default. The width of the parallel container will be the same as the length of the longest of the parallelized invocations. Figure 10 shows the prior example as it would appear in the 2D profile.

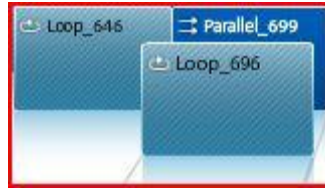


Figure 10. Default view of a parallel invocation in the 2D profile

If you select one of the shorter parallelized invocations in the 1D profile, that invocation will be shown as the child in the 2D profile as well, but will, of course, be shorter than the parallel container itself. Figure 11 shows the prior example, but now the shortest of the three parallel invocations selected and displayed. In the 2D profile, *Loop\_697* is clearly shorter than its container *Parallel\_699*.



Figure 11. Parallel invocation with shorter loop selected

### Intrinsics and idle time

Part of the job of optimizing a parallel implementation involves making sure all processors are busy all the time. Being able to view idle time is useful so that operations can be rebalanced, with idle time reduced as much as possible.

Idle time arises from one of two sources:

- some operation outside the program itself has been requested and the results are pending
- execution of some code has progressed to a point where it cannot proceed until some other task has finished a computation.

Examples of the former are I/O operations. Examples of the latter are when forked operations join, with one of the execution branches completing ahead of the others, or an unbalanced pipeline, where one stage is faster than another and has to wait.

Both of these involve the use of *intrinsics*; intrinsics are operations that aren't a primitive language expression, but whose operation is opaque (meaning you have no visibility into what's going on). These are typically library calls. Both sources of idle time reflect the use of intrinsics in a program, either for I/O or for synchronization. An intrinsic is depicted in the

vfAnalyst display as shown in Figure 12. Intrinsic can be filtered from the 1D display to remove clutter.



Figure 12. An intrinsic.

Idle time will be visible in a subsequent release of vfAnalyst.

### Branching constructs

It's typical for control structures in a program to provide for alternative execution paths. These would typically be through *if/then/else* types of statements or *case* or *switch* statements. These are generically referred to as branching constructs or conditionals. Branching constructs are not invocations and don't appear in the 1-D profile, but they are shown in the 2-D profile.

A branching construct is depicted by vfAnalyst as shown in Figure 13. The area bounded by the white box indicates that portion of the function covered by the branching construct, with a line dividing the alternative branches of execution. The width of a branch section is proportional to the contribution of that branch to the overall delay of the branch construct as a whole.

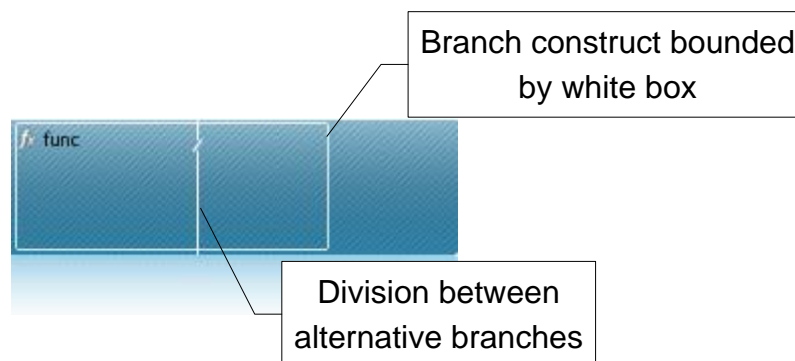


Figure 13. A branching construct

### **Dependencies**

The key to effective parallelization of sequential code is dependencies. There are different types of dependencies, and what follows is a discussion of each type, along with an illustration of how they are depicted in vfAnalyst. It's possible to use vfAnalyst to parallelize your code without understanding the dependencies, but if you're not satisfied with the partitioning options that vfAnalyst gives you, understanding the dependencies will help you see whether there are issues blocking parallelization. You may be able to use this information

to improve your code and make it easier to parallelize. The process of parallelizing code is discussed in a separate parallelization whitepaper.

### Compute dependencies

If one calculation uses the results of another calculation, then it must wait until that prior calculation is complete before it can proceed. This is referred to as a *compute dependency* because it is driven by the computing flow of the program. Within the scope of vfAnalyst's view, where key events are memory loads and stores, a compute dependency starts with a *load* operation, retrieving a value from memory. That value is then operated on until the result is placed back in memory, ending in a *store* operation. This type of dependency is readily identified through static analysis performed by vfAnalyst. In vfAnalyst, a compute dependency is depicted as shown in Figure 14.



**Figure 14. Compute dependency**

Note that a *load* operation is represented by an upwards-pointing triangle; a *store* operation by a downwards-pointing triangle. Any arrows are always in the direction of data consumption.

### Memory dependencies

The next type of dependency is less obvious. It is made up of the hard-to-track activity primarily associated with pointers. The C language in particular allows undisciplined, unstructured use of pointers in a manner that may be by turns considered sloppy or clever. They are the single biggest source of unanticipated, hard-to-find, subtle bugs and problems, especially when trying to parallelize a sequential program. They cannot be handled by compilers or standard profilers.

The standard “solution” to dealing with pointers is to forbid them, which places a huge rewriting burden on the engineer, who may not have even been the original author of the code. Rather than forbidding pointers, vfAnalyst can analyze their behavior and identify the critical dependencies that they create. This eliminates the need to remove pointers to parallelize a program.

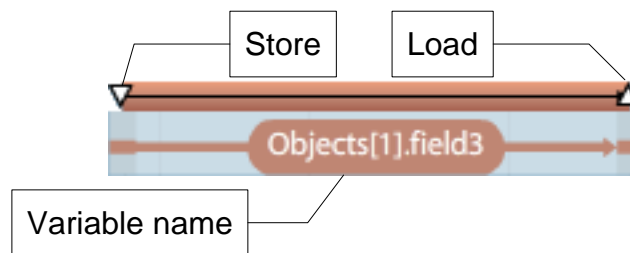
Because of their association with pointers and memory access, these dependencies are referred to as *memory dependencies*. They start with a *store* operation, which is typically (but not exclusively) a pointer write, and end with one or more *load* operations, which are typically pointer references.

What makes these dependencies not statically analyzable is the fact that there may be many pointers referencing the same address at any given time during program execution, each of

which has a different name. As a result, dynamic analysis is needed in order to identify what is reading from and writing to specific addresses.

If a program is parallelized without correctly accounting for these dependencies, behavior will be badly broken in ways that may be very hard to debug. Because this storing of data for later loading is effectively the communication of data from one part of the program to another, incorrect parallelization means that some parts of the program will not have the correct data. vfAnalyst's ability to deal with memory dependencies is unique and paramount.

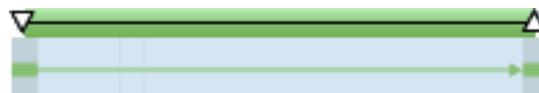
Memory dependencies are depicted in vfAnalyst as shown in Figure 15.



**Figure 15. Memory dependency**

### Streaming Patterns

One of the key differentiating characteristics of vfAnalyst is its ability to detect patterns of behavior. Streaming data communications are particularly important for identifying good places to parallelize code. Streaming patterns have a special representation in the 2D-profile as shown in Figure 16.



**Figure 16. A streaming data pattern**

The list of which streaming patterns can be detected will grow over time; the current list is summarized below. These patterns are described in more detail in the [Recognized Patterns document](#).

| <b>Pattern</b> | <b>Description</b>   |
|----------------|--|
| FIFO           | Indicates that each data item stored is loaded exactly once, and that the loads occur in the same order as the stores.   |
| Windowed FIFO  | Indicates that data items are written and read within a range of addresses; the range increases monotonically, but, within the range, the stores and loads can occur in any order, multiple times or not at all. |

### Anti-dependencies

The dependencies discussed so far hinge on the need for data to be available for use, which means waiting to read the data. The reverse situation can also be true: a piece of data that is going to be overwritten may have to hold its current value until that value is no longer needed. This means delaying a write until all required reads are complete. These are referred to as *anti-dependencies*.

Anti-dependencies are depicted just as shown in Figure 17.



Figure 17. Anti-dependency

### Invocation entry/exit points

Some variable loads and stores occur at the boundaries of invocations. In the case of functions, loads may be accomplished through the use of function parameters; likewise, some variable stores occur as function return values. In the case of loops, there may be stores and loads associated with a loop entry variable that changes on each iteration or a computation result from the loop. Such loads and stores are indicated by the use of brackets at one or both ends of the dependency, as shown in Figure 18.



Figure 18. Invocation entry and exit points

### Out-of-program dependencies and initialization

All programs rely on some implicit initialization before running. There may be dependencies between explicit activity in the program and this implicit initialization. Such dependencies are depicted as shown in Figure 19. Note the lighter gray element on the far left side of the linear view; this represents activity that occurs immediately before explicit code begins execution. In the example shown, variable *x* is accessed at some point, and expects an initialized value.

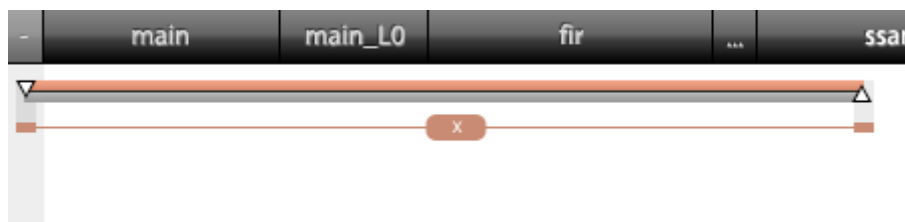
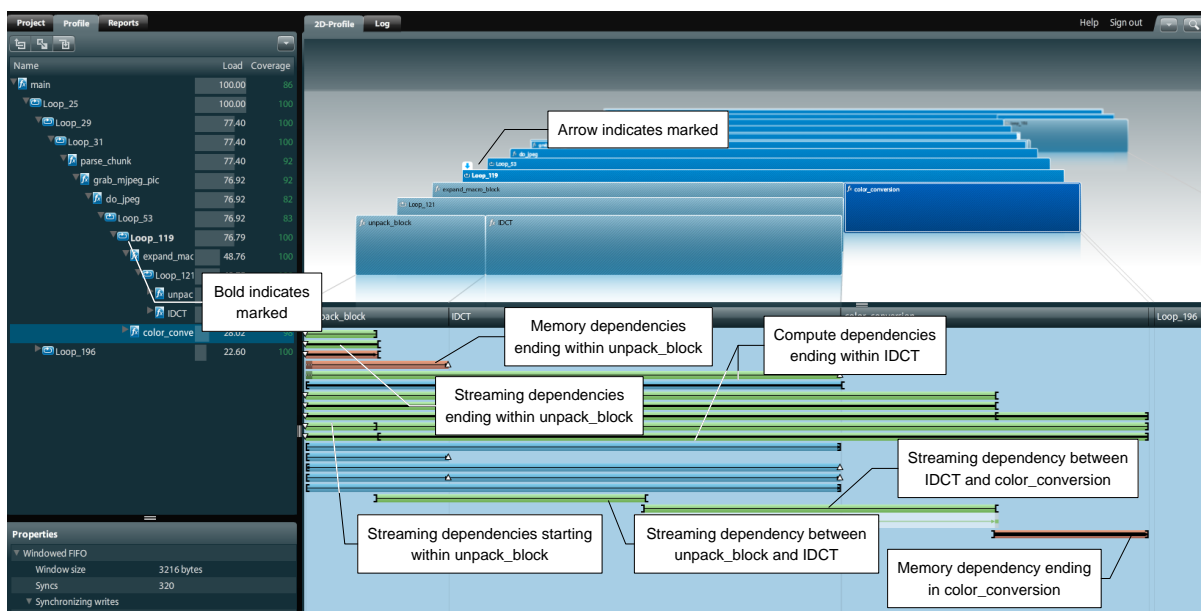


Figure 19. Initialization dependency

## Selecting which dependencies to view

Most programs have large numbers of dependencies, so simply displaying all dependencies results in too much information to be useful. Instead, vfAnalyst allow you to determine which dependencies will be shown.

In the first releases, you will be able to *mark* one invocation in the 1-D profile. Dependencies between the marked invocation and all invocations below it in the hierarchy will be visible. You can hide some dependencies by collapsing invocations; any dependencies wholly contained within a collapsed invocation will not be visible. An example is shown in Figure 20.



**Figure 20. Selective dependency display**

You can also reduce screen clutter by suppressing the display of different kinds of dependencies. For example, anti-dependencies are currently suppressed by default.

Because you can limit the scope of dependency viewing, you may have a dependency that “touches” an invocation that is visible (perhaps there is a read in the invocation), but also touches invocations that aren’t in view (perhaps the write for the dependency is elsewhere, or there is another read somewhere else). In this case, vfAnalyst also displays brackets at the end of the visible portion of the dependency.

Figure 21 shows several streaming dependencies that have brackets on the right. That’s because the dependencies actually end in function *process\_MCU()*, but only *Loop\_585*, contained in *process\_MCU()*, is selected for dependencies (as shown by the blue downward arrow), meaning that then ends of those dependencies are outside the scope of the current view. Selecting *process\_MCU()* for dependencies would reveal the actual dependency ends. That information is also available in the dependency properties.

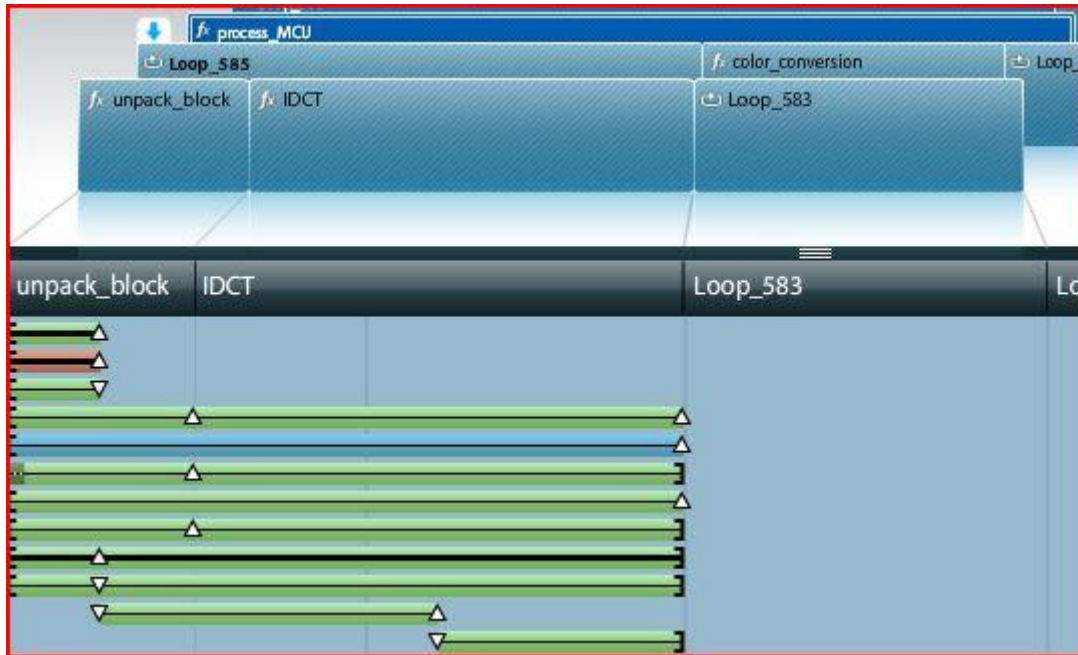


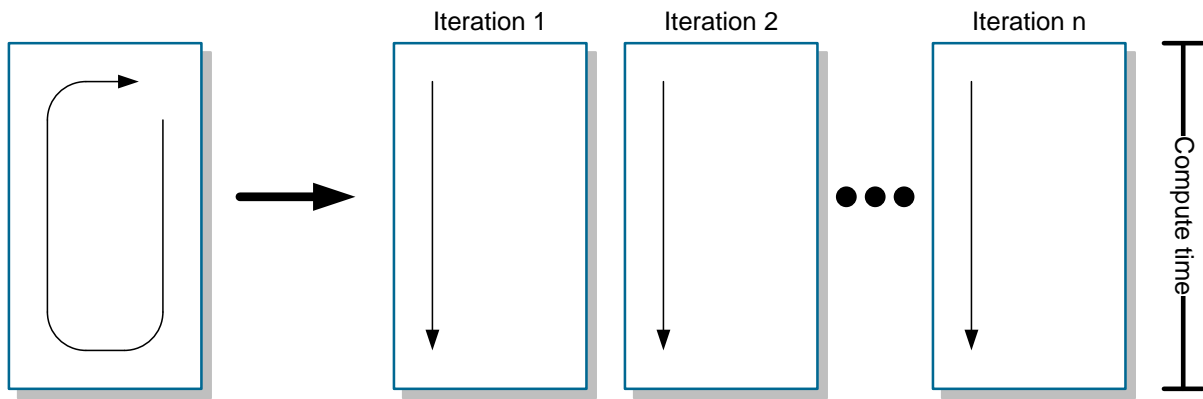
Figure 21. Dependencies that extend beyond the current view

### Loop-carried dependencies

There's a particularly important kind of memory dependency or anti-dependency: it's the consumption in one iteration of a loop of some value produced in another iteration. An example would be the equation,  $a[i]=a[i-1]+b*a[i-2]$ . Array notation is used here because loops are so commonly used to traverse arrays. And here the notion of traversing an array becomes bound up with the notion of time, and the notion of addressing the array can be confused with the notion of which iteration produced the data; we'll deal with that shortly.

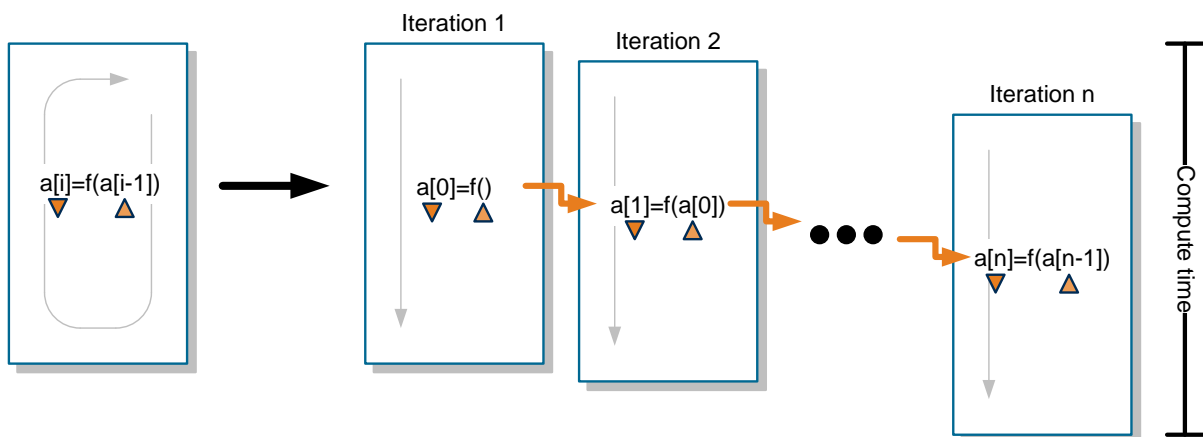
When a loop is used, with one iteration executed for each array element, each iteration thinks of  $a[i]$  as the "current" iteration, with earlier elements belonging to prior iterations. When an element in one iteration consumes an element produced in another iteration, it's referred to as a *loop-carried dependency*. These dependencies are important enough that we will spend extra time describing them.

Loops can often be parallelized to improve performance. This means that each iteration of a loop can be assigned to a different task. If there are no dependencies between the iterations, then each iteration can happen at exactly the same time, as shown in Figure 22.



**Figure 22. Unrolled loop with independent iterations**

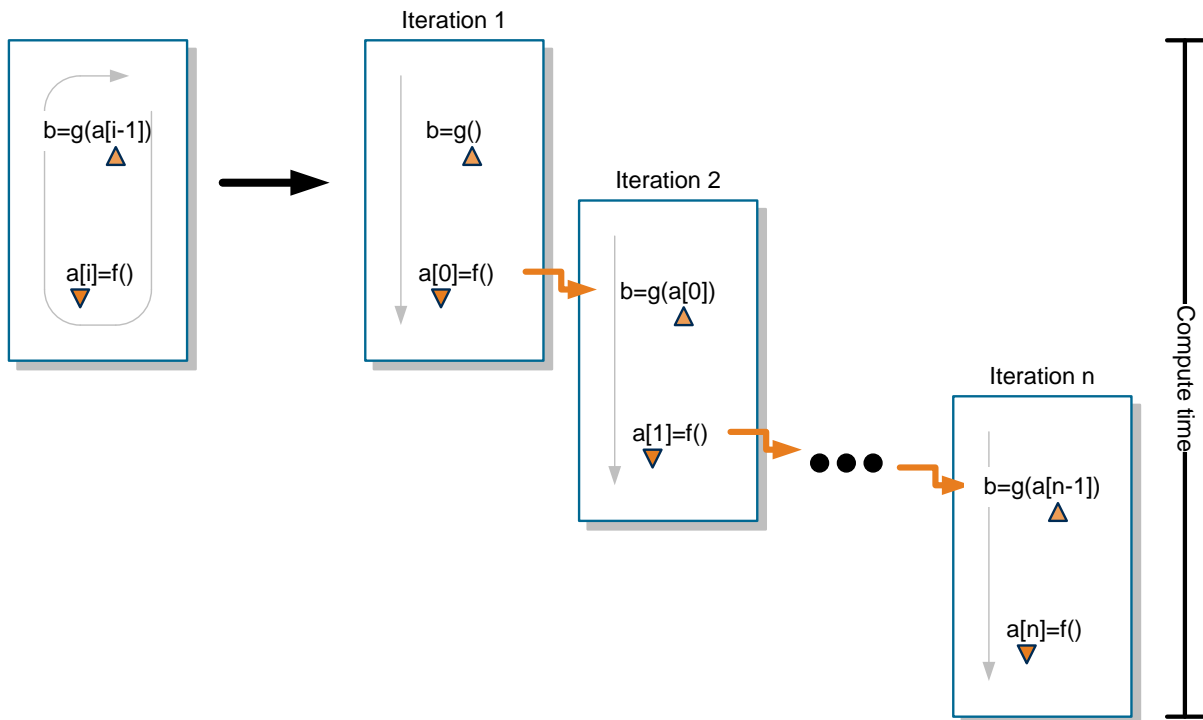
If there are loop-carried dependencies, some adjustments are required. The simplest and best case is one where the assignment of a new value depends on its prior value – say,  $a[i]=f(a[i-1])$ . The reason this is simple is that the statement that produces  $a[i]$  is also the consumer of the prior iteration’s data  $a[i-1]$ . Because these occur so close together, the consuming event doesn’t have to wait for long to get the produced data. Figure 23 shows this.



**Figure 23. Unrolled loop with nearby loop-carried dependency**

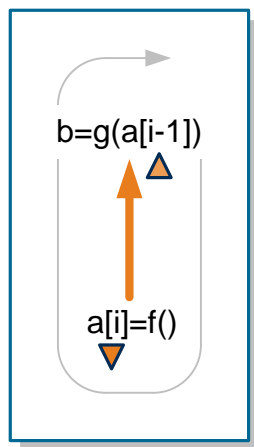
We’ll ignore boundary conditions at the beginning of the loop for this discussion (which is why  $f()$  has no detail in the first iteration). In the second iteration, the computation of  $a[1]$  requires  $a[0]$ ; therefore that has to wait until  $a[0]$  is available, as shown by the orange arrow. Because the production and consumption of these values of  $a$  are so close together, there is little penalty. The entire loop does take longer than the independent loop shown in Figure 22, but not by much.

On the other hand, if the consumption of the data comes early in the loop but the production comes late, the delay can be more significant, as illustrated in Figure 24.



**Figure 24. Loop-carried dependency with greater delay**

Because, in this case, the data is being produced late in the loop body and consumed early in the loop body (although in a different iteration), this dependency is represented as a backwards arrow (since the arrows always go in the order of data consumption); Figure 25 motivates that, but we'll look at an actual vfAnalyst depiction shortly. Note that such a dependency can occur with a forward arrow, but in that case, because the data is produced before it's consumed, it doesn't delay the loop instances when unrolled.

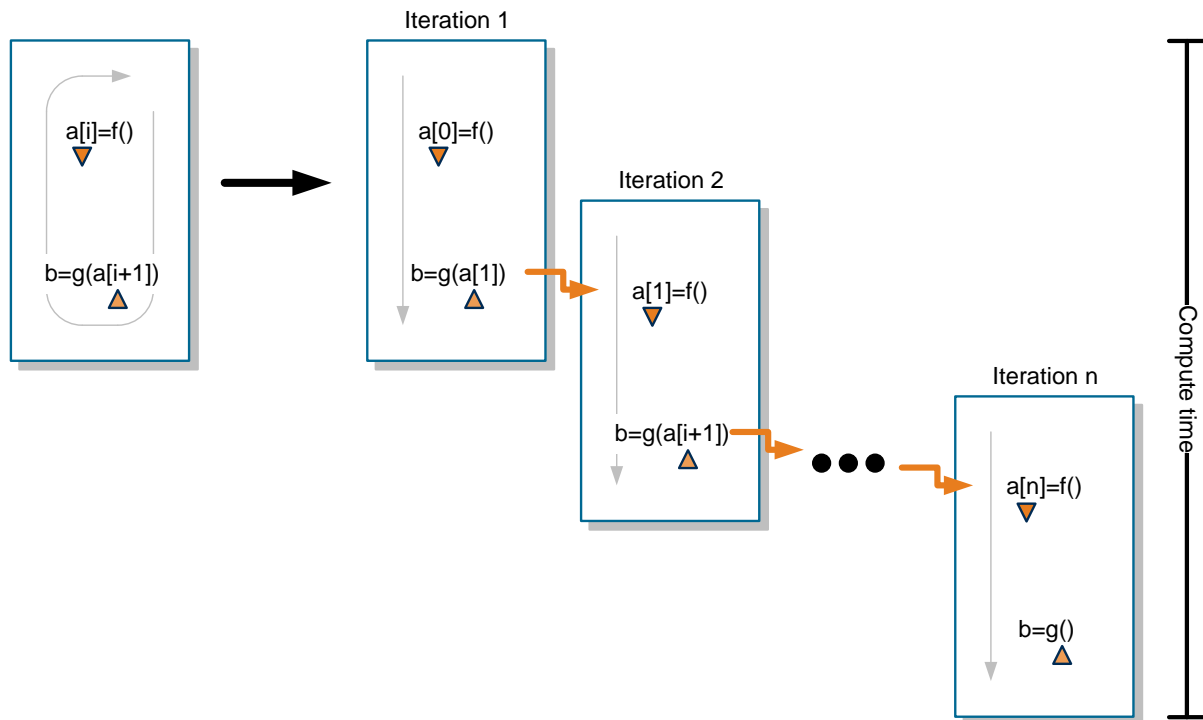


**Figure 25. Loop-carried dependency has a reverse arrow**

Note that the “position” of these statements with respect to each other, strictly speaking, is not directly tied to the order of the statements as typed into the source code: it's tied to the order in which they execute. In theory, the compiler may reorder statements to improve performance, making actual execution order different from typed statement order. In practice,

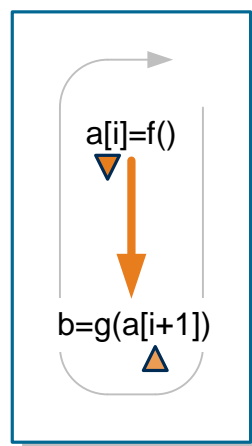
however, such re-ordering tends not to be extensive, so the executed order is likely to be similar to, if not exactly the same as, the typed statements.

Loop-carried anti-dependencies behave similarly, only now the production of new data has to await the consumption of old data. The effect is the same if the production and consumption are far apart, as seen in Figure 26.



**Figure 26. Loop-carried anti-dependencies**

Here the producing statement comes ahead of the consuming statement, only in a later iteration. Therefore a forward arrow would be indicated, as shown in Figure 27. If the data were consumed ahead of production, the arrow would go backwards, but it wouldn't cause a delay in the loop iterations when unrolling and so is of less interest.



**Figure 27. Loop-carried anti-dependency has a forward arrow**

### The dependency distance

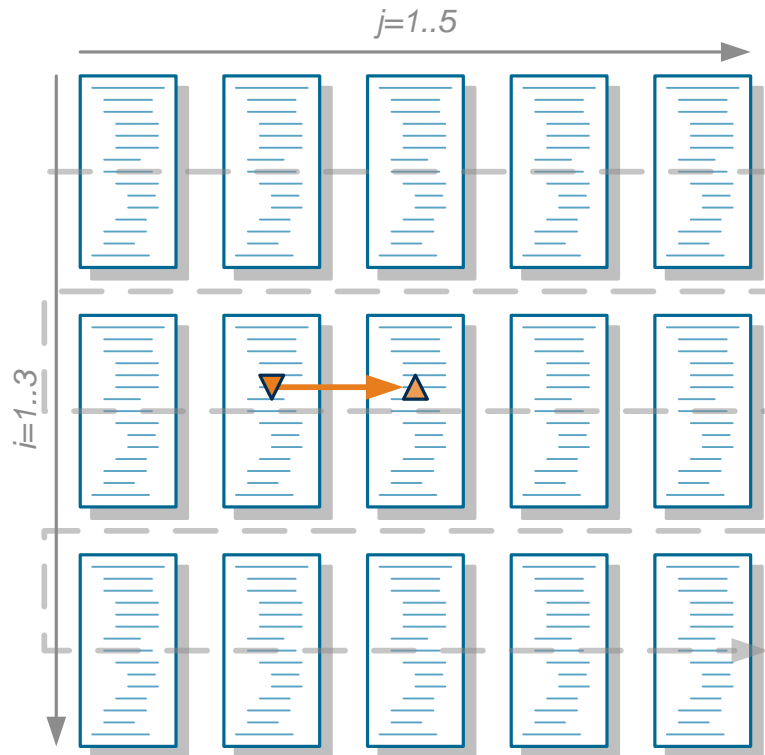
The direction of data production and consumption within an iteration (forward or backward) provides no information about how many iterations we have to reach across to get data. Even the arrow direction is insufficient, since we can have forward and backward arrows for dependencies and anti-dependencies. The remaining information is provided by the *distance*, which is the number of iterations between data production and consumption. The distance will always be a positive number. So if  $a[0]$  is used to calculate  $a[1]$  (meaning that the formula for  $a[i]$  depends on  $a[i-1]$ ), then data is produced in one iteration and consumed one iteration later, and so the distance is 1.

Loop nesting makes the distance more complicated. Let's start simple with a two-level nested loop (using simplified pseudo-code):

```
for i=1..3
  for j=1..5
    a[j]=x*a[j-1]
```

In all of these examples we'll ignore boundary conditions since they're not relevant to the discussion.

The above example shows that calculating the value  $a$  for the current loop iteration ( $j$ ) requires the value from the prior loop iteration ( $j-1$ ). This may be easier to visualize by drawing out all of the iterations that will occur for both loops as shown in Figure 28. Note that this drawing doesn't depict data in an array, but rather complete iterations of code as the loops progress.



**Figure 28. A dependency on the prior iteration**

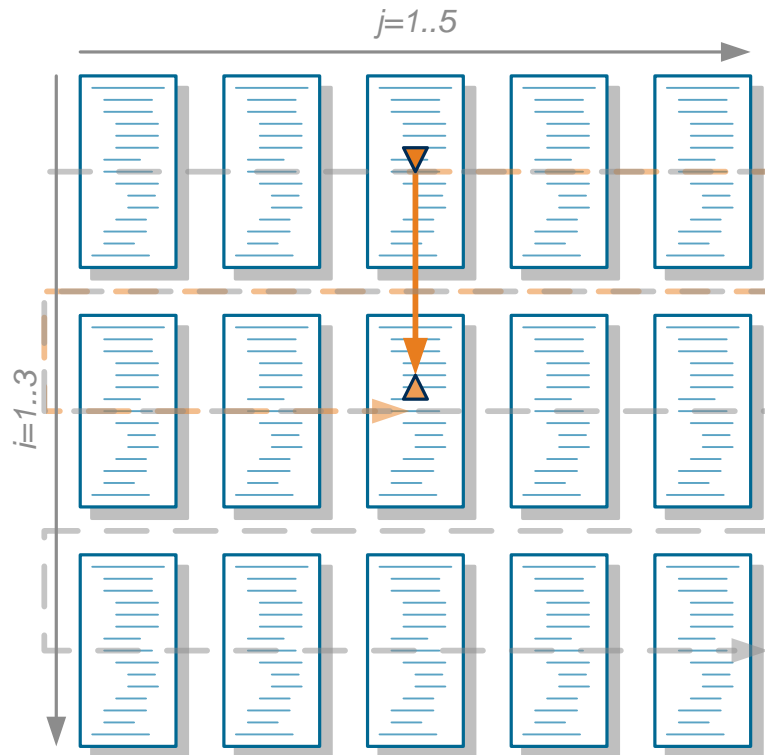
Here we're showing an arbitrary instance, the 8<sup>th</sup> one, that receives data calculated in the 7<sup>th</sup> iteration, as indicated by the arrow (where the arrow always goes from producer to consumer). Because the data "travels" one iteration between production and consumption, the distance is 1. But things can get more complicated, as indicated by the following loop statement, depicted in Figure 29.

```

for i=1..3
  for j=1..5
    a [j]=x*a[j]

```

The implications of this may not be immediately obvious, and so bear some more explanation. Here a new  $a[j]$  is being calculated based on the current value of  $a[j]$ . The current value of  $a[j]$  is the one that was calculated on the prior iteration of the outer  $i$  loop while  $j$  was at its current value. Therefore we're reaching back one iteration of the outer loop for this data.



**Figure 29. Dependency on prior outer iteration**

In this case, the data used in the 8<sup>th</sup> iteration (which is the 3<sup>rd</sup> inner loop iteration of the 2<sup>nd</sup> outer loop iteration) comes from the 3<sup>rd</sup> iteration (which is the 3<sup>rd</sup> inner loop iteration of the 1<sup>st</sup> outer loop iteration). So the distance is 5.

But this distance can also be visualized as a *distance vector*, with an index for each loop. In this example, we had to reach back one outer loop iteration to get the data, but it was the same inner loop iteration. So the distance vector is  $[Li=1, Lj=0]$ .

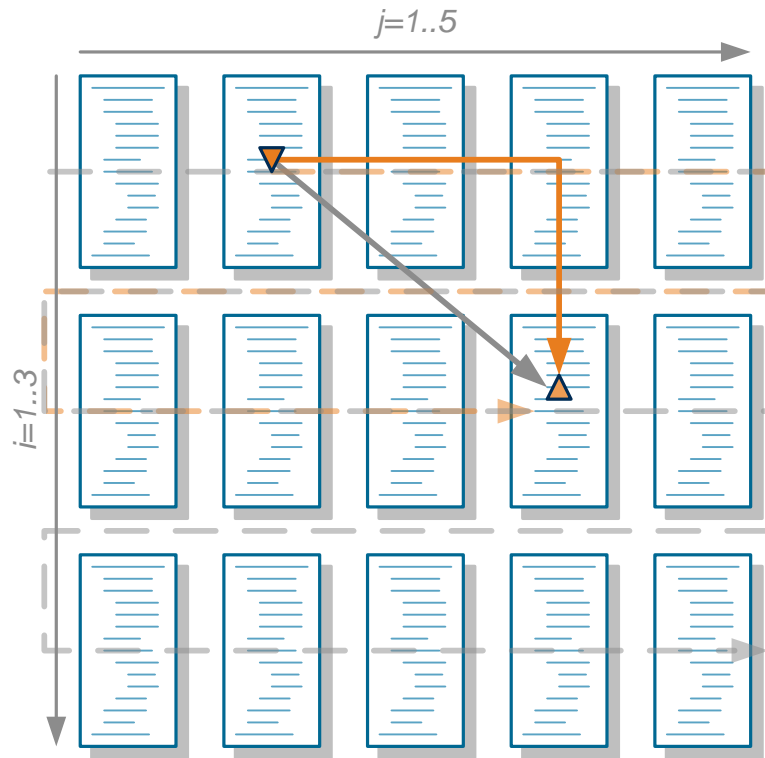
The overall *scalar distance* is calculated according to the loop limits. The inner loop has five iterations for each outer loop iteration. Our distance vector specifies one outer loop iteration and no inner loop iterations, so we have  $5*1 + 0*0 = 5$ , as we observed visually above.

We can show this in a slightly more complicated case, as depicted in Figure 30.

```

for i=1..3
  for j=1..5
    a[i][j]=x*a[i-1][j-2]

```



**Figure 30. Dependency on prior inner and outer loop iterations**

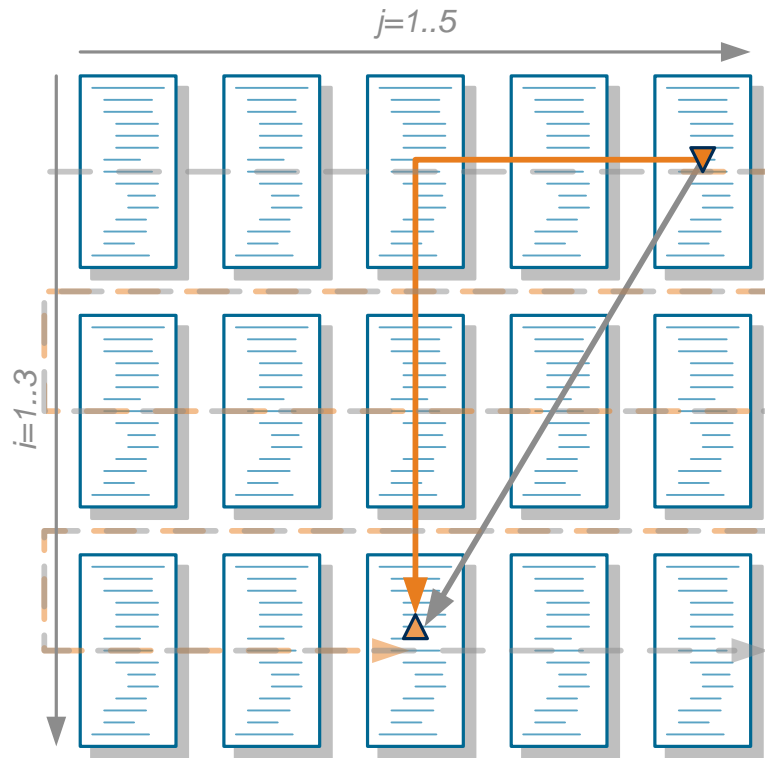
Here we are reaching back not just one outer loop iteration, but also two inner loop iterations. The distance vector is  $[Li=1, Lj=2]$ , which makes the scalar distance  $1*5 + 2$ , or 7, as indicated by the dashed line.

Finally, it is possible for any except the leftmost component of the distance vector to be negative. An example of this is shown in Figure 31

```

for i=1..3
  for j=1..5
    a[i][j]=x*a[i-2][j+2]

```



**Figure 31. Loop-carried dependency with a negative vector component**

Here we reach back two outer loop iterations, but it looks like we're reaching forward on the inner loop. We can do this because it's a reference to a prior outer-loop iteration, so it still reflects something that happened in the past. Here the distance vector is  $[Li=2, Lj=-2]$ , and the scalar distance is  $2*5 - 2 = 8$ .

If you just look at some of the examples above, it's easy to confuse the iterations with actual array addressing. The third and fourth examples might lure you into such thinking, but the second example then becomes exceedingly confounding. Even so, these have been extremely trivial examples; they can be much more complex and convoluted, and trying to sort out the distance can be vexing if you're not an expert embedded in this world. That's where vfAnalyst helps out: you don't have to figure it out; vfAnalyst will do it for you. All you need to do is take the result and make decisions.

The scalar distance is useful because it tells you how many instances the data producer can run in parallel. It also gives you a general sense about your opportunities for exploiting concurrency: the longer the distance, the more time you have between production and consumption, allowing for a slower and, presumably, less expensive communication mechanism.

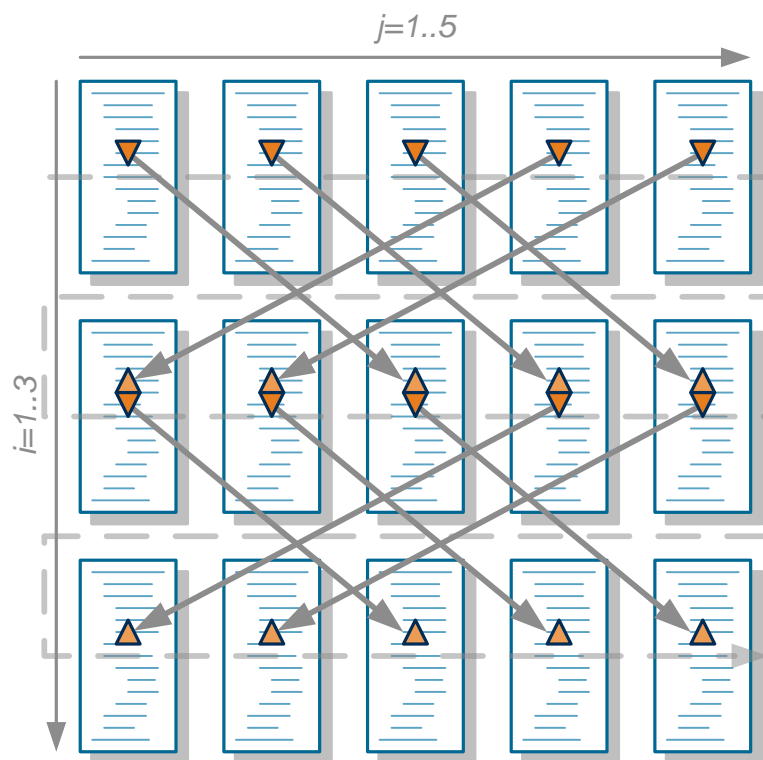
The distance vector gives you an indication as to where convenient synchronization points might be. These are the points at which the producer stops writing data and turns it over to the consumer to read. They introduce overhead, so synchronizing after each write of data may be too expensive; on the other hand, waiting until all the data is written doesn't allow reading processes to proceed in parallel. Introducing periodic synchronization (or *sync*) points allows

transfer of data with less overhead. So, for example, if two nested loops have a distance vector of [5;10], a total of 50 iterations occur, but the inner loop runs 10 times for each iteration of the outer loop. Thus it may be convenient to synchronize every 10 iterations.

### Store and load occurrences

These loop-carried dependencies arise from store operations in one iteration (or before the first iteration) and load operations in a subsequent iteration. But, while the dependencies may exist as a whole, they may not exist for every operation. For example, if there is an if/then/else clause in the code, then, depending on the condition being tested, either the store or the load may not occur in any given iteration.

vfAnalyst can tell you which iterations of a nested loop caused stores and loads to occur. As an example, in the case of the dependency illustrated in Figure 30, there would at minimum be some code testing to ensure that no attempt was made to reach outside the array bounds. So the stores and loads would occur as shown in Figure 32.



**Figure 32. Store and load occurrences**

You can see here that stores happen in each  $j$  iteration, but only in the first two  $i$  iterations. vfAnalyst will report the store occurrences therefore as  $[Li=1..2, Lj=^*]$ , where the  $*$  indicates all iterations.

Likewise, load the load occurrences are  $[Li=2..3, Lj=^*]$ .

This is another of those situations where the tool will report only the behavior it sees. If the test data is such that certain branches of code aren't covered, then vfAnalyst will not be able to report the missing behavior. When looking at the store and load occurrences, if what you

see surprises you, then the first thing to do is confirm that your test set is complete. If not, then add the necessary data and re-run the analysis. If it is complete, then vfAnalyst has told you something about your program that wasn't obvious to you without it.

### Loop-carried dependencies in the vfAnalyst display

vfAnalyst provides all of the information necessary for understanding the loop-carried dependencies in a program. An example is shown in Figure 33. Here there is a memory dependency with variable `Objects[1].field3`; that variable can be clicked in the properties pane to get to the source code and see the original name and context.

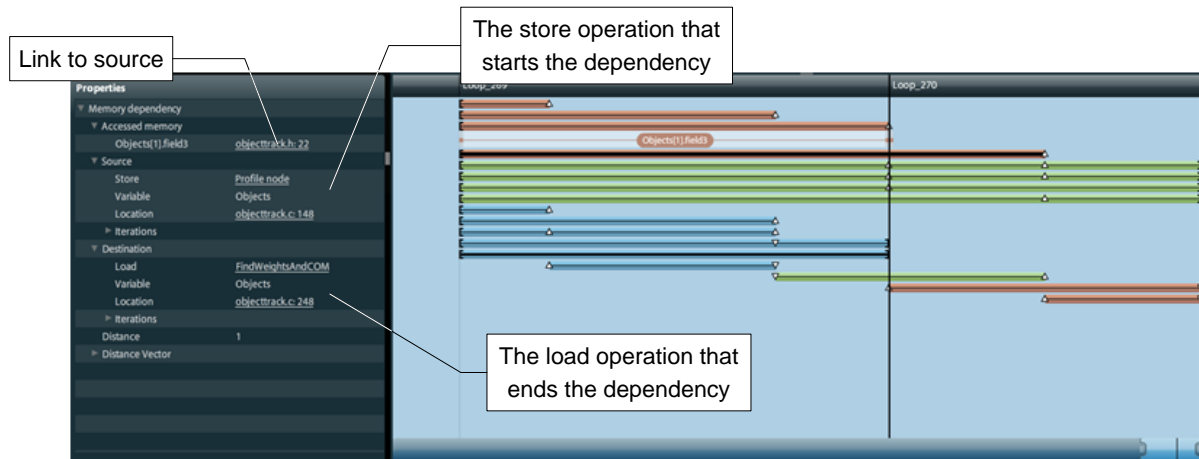


Figure 33. Nested loops with a loop-carried dependency

### Induction variables and expressions

There is a special category of expression within a loop referred to as an *induction expression*. The variable being assigned in the expression is an *induction variable*. Induction expressions have the following specific characteristics:

- Their next assigned value of the induction variable is computed only from one or more prior assigned values and constants (that is, no other variables).
- They require no memory accesses.
- They are directly computable in the sense that, given the expression and an initial condition for the first iteration, the expression can be transformed in a way that makes it possible to calculate the value of the induction variable for the  $n^{\text{th}}$  iteration directly, without actually iterating through all  $n$  iterations to determine the result. In practice, this limits the range of possible operations to relatively simple ones – like linear expressions – and limits the range of prior values that can be used.

The classic example of an induction expression reflects incrementing the loop *iterator* (archetypically referred to as  $i$ ):

$$i = i + 1$$

(which, in terms of current and prior values, is  $i_n = i_{n-1} + 1$ ). On the other hand, the deceptively simple-looking Fibonacci expression

$$x_i = x_{i-1} + x_{i-2}$$

does meet the condition, but only barely – the calculation required is complex. The expression

$$x_i = x_{i-1}^2 + 1$$

would not be a valid induction expression. It is not obvious from inspection which expressions are induction expressions, but vfAnalyst can recognize them.

Induction expressions are useful in setting initial conditions when loops are parallelized. Therefore vfAnalyst reports them separately as a property of the loop in which they're found. The loop example of Figure 3 is repeated here as Figure 34 to illustrate the induction expression information. In this example, the induction variable is called *p63*, and the induction expression is

$$p63 = p63 + 1.$$

| Properties              |                              |
|-------------------------|------------------------------|
| ▼ Loop_269              |                              |
| Loop                    | Loop_269 (FindWeightsAndCOM) |
| Compute load            | 3.6 %                        |
| Line coverage           | 100.0 %                      |
| Iteration count         | 4096                         |
| Source location         | objecttrack.c: 236-240       |
| ▼ Induction Expressions |                              |
| indvar35 = indvar35 + 1 | objecttrack.c: 236           |

Expression with machine-generated induction variable name

Clickable location of original induction expression

**Figure 34. Induction expressions**

### Variable Aliasing

Note that, for this example above, *p63* is not the actual variable name from the original code. This is an example of *variable aliasing*, and it can happen in a couple different ways.

The most obvious case happens when some memory location is passed as a parameter to a function. This is more commonly known as *pointer aliasing*. The original variable might be named *x*, but the parameter declaration may be named *k*. When the function executes, *x* is passed as an argument, but during execution it appears that *k* is the name of the variable being operated on. At this time, *x* and *k* refer to the same memory location. On a different invocation of that same function, with a different parameter value, *x* and *k* may no longer refer to the same memory location.

When this kind of aliasing occurs, vfAnalyst tries to correlate the alias to ensure that both the alias and original names are reported.

In a more subtle case, compiler optimizations can change the apparent name of the variable. It may be harder in this case for vfAnalyst to correlate the internally-generated variable name – such as the *p63* above – to the original name. While this capability will improve with subsequent releases of the tool, clicking through to the source code will always help identify what the original variable name is.

### Conclusion

The ways in which a program behaves, and, most importantly, the ways in which different parts of a program depend on each other, play a large part in the decisions that must be made when attempting to parallelize a program. In this paper, we have outlined the important concepts that such behavioral analysis comprises, using the display of vfAnalyst as a means of illustrating a practical implementation of such analysis.

Armed with such analysis results, programmers will be better equipped to create parallel implementations of their programs. A separate paper will explore the ways in which this information can be leveraged.