Integrating the Hardware Description With Gedae's Single Sample Language to Generate Efficient Code

William Lundgren, James Steed, Kerry Barnes
Gedae, Inc.
18000 Horizon Way, Suite 200
Mount Laurel, NJ (USA) 08054

Abstract

Gedae is a powerful tool for creating deployed systems and advanced demonstrators for boards of digital signal processors (DSP) or distributed networks. To create efficient programs, the tool depends on the presence of a C cross compiler and optimized libraries for each processor type it targets. For example, Gedae performs a vector add efficiently because the vendor (or a third party) supplies an optimized vector add routine which can be called from Ansi C-code generated by Gedae. To minimize this dependence, a project has been started to develop concepts for using Gedae's single sample language, Gedae-RTL, to export efficient, target-optimized code in any programming language. To accomplish this extension, the hardware description in Gedae's embedded configuration will be extended and tightly integrated with the Gedae-RTL code generation. This paper will discuss how the hardware description and Gedae-RTL will be extended and integrated to accomplish these optimizations.

Keywords: Gedae, target optimization, rapid development, code generation

Introduction

Gedae is an integrated design environment for deployed systems and advanced demonstrators based on boards of digital signal processors (DSP) (e.g., AltiVec, PowerPC, TigerSHARC) or distributed networks (e.g., Linux clusters). The core Gedae programming language is based on streams of data, where primitives process buffers in a queue that can contain large amounts of data. Increasingly, field programmable gate arrays (FPGAs) are being used alongside DSPs as a method for meeting high data flow requirements. With the increased focus on targets such as FPGAs, a single sample meta-language, Gedae-RTL, has been added to Gedae to allow mapping to these devices. While core Gedae primitives are written in Ansi-C (with Gedae-specific extensions), Gedae-RTL can be used to export code in any programming language, whether it's the same generic Ansi-C or VHDL for generating firmware. The language can be used for much more than programming FPGAs, as the functional depiction provided by the language can be just as closely tied to a fixed architecture such as a DSP as it can to a customizable one such as FPGAs.

This language independence is provided through the Language Support Package (LSP). The LSP is a set of functions which translates an internal netlist data structure into code which can be compiled by the vendor tools. No longer being tied to generic C-code, this language independence provided by the LSP creates an opportunity for target-based optimizations outside of those provided by vendor libraries. To

optimize Gedae-RTL graphs based on the target, Gedae must have knowledge of the architecture. The embedded configuration file lists a high level view of the architecture, giving the processors types and describing their interconnections. This hardware description must be extended to describe the architecture at the processor level. While these developments are also applicable to FPGAs, the focus of this paper will be how the integration of Gedae-RTL with a more detailed hardware description will be used to improve the efficiency of applications for fixed architectures, such as DSPs and workstations.

Single Sample Language

Gedae has been augmented with a single sample meta-language based on the theory of register transfer languages called Gedae-RTL. This language is capable of exporting VHDL code for FPGAs as well as Ansi-C code optimized for a DSP.

Functionality built using Gedae-RTL uses the new single sample primitive type. Conceptually, a graph of single sample primitives forms a processing pipeline that is driven by a clock. These single sample primitives are built upon seven fundamental functions:

- Register: `R(in,out,c)`
  On a pulse from clock `c`, the value of `in` is copied to `out`, creating a delay of one clock period. The R function is the only method to retain state data in a Gedae-RTL graph.
- Assignment: `A(E,out)`
  Evaluate the expression `E` and assign its value to `out`. Example: `A(a+b,out);`
- Decimate: `D(in,c)`
  Tie clock rate `c` to signal `in`. The value of `in` is only computed when `c` is true. The D function is different from the R function in that there is no retention of state and there is no delay in the availability of the result.
- Clock: `C(in,c)`
  Get clock rate `c` tied to `in`. This function is the inverse of the D function and provides a convenient way of accessing the clock rate of a signal to use on other signals in close proximity.
- Memory: `M(in,n,s)`
  Allocate buffer `in` with `n` elements of size `s`. This allocation can be mapped to a specific memory either with an optional fourth parameter or in the implementation dialogs.
- Read: `MR(a,out)`
  Read the element at address `a` and put the value in `out`.
- Write: `MW(in,a)`
  Write the value `in` to address `a`.

Much like Gedae's core language, the Gedae-RTL graph specifies only the functionality of the graph without regard to the target or its programming language. For example, Figure 1 shows a low pass filter implemented in Gedae-RTL (`out(i) = out(i-1) + K*(in(i)-out(i-1))`), built from a register (`reg`), a multiplier (`mult`), and two adders (`add` and `sub`) with no target-specific processing. Through the LSP, target code is exported to implement the application, and Ansi-C code is created for simulation. Components implemented in Gedae-RTL interact seamlessly with core Gedae components. The language processes scalar tokens, where vector and matrix tokens are considered to be memory allocations that must be accessed via the MR and MW functions.

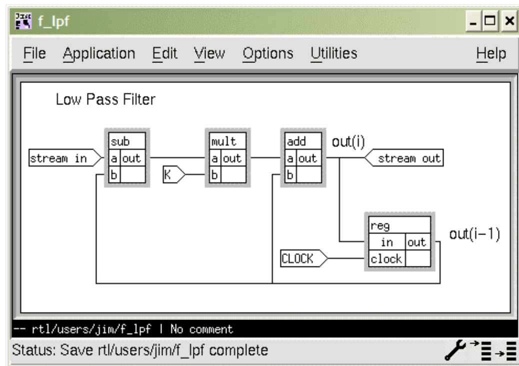


Figure 1 – Low pass filter implemented in Gedae-RTL

Language Support Package

Much like Gedae's core language, the Gedae-RTL graph specifies only the functionality of the graph without regard to the target or its programming language. For example, Figure 1 shows a low pass filter implemented in Gedae-RTL. The graph in Figure 1 includes only multipliers, adders, and delay registers. In other words, the graph is not specific to FPGAs or firmware; it is simply a specification of a low pass filter. Gedae is able to export code in potentially any language to implement the functionality specified in the graph.

To support any language, the Language Support Package (LSP) has been integrated into the Gedae-RTL code generation process. The Gedae-RTL graph is not translated directly into code. Instead it is transformed into an internal netlist representation and information on the algorithm is collated in data structures. Implementation settings from the graph developer also affect the internal data structure, allowing the developer to insert registers and map memories to hardware components to enhance the implementation. Once the internal implementation is created, the LSP uses the information provided by Gedae to export code in the target language. The target language could be VHDL for an FPGA, assembly code for another alternate architecture, or Ansi-C code with DSP-specific enhancements. The code is then built by Gedae's customizable make system.

The architectures that can be targeted by Gedae-RTL through the LSP covers the entire range of types of hardware. On one end, FPGAs provide a highly customizable target where the graph can simply be translated into a circuit and implemented in the slices of the FPGA. If the synthesis constraints require a high throughput, the Gedae-RTL depiction can be automatically altered, under the control of the user, through the insertion of registers to reduce the settling time. On the other end of the spectrum, DSPs provide little build-time customizability. However, the direct specification of registers and the breadth of port information make the Gedae-RTL depiction a very powerful means of specifying an algorithm so that it can be optimized for the DSP hardware. The synchronization of the algorithm can be analyzed by Gedae, and the graph can be modified by Gedae to produce an order of operations suited to the target architecture.

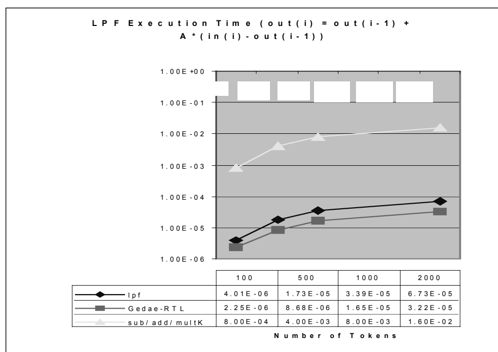
Efficiency of Generated Ansi-C Code

An investigation of the efficiency of Ansi-C code generated by Gedae-RTL and compiled by vendor tools provides the motivation for the need for target-based optimizations. To test the efficiency, three benchmarks are investigated, MMA (`out = a*b + c*d`), low pass filter (`out(i) = out(i-1) + K*(in(i)-out(i-1))`), and edge detection (3x3 Sobel kernel operation), and the results are compared against primitives from Gedae's core library which in most cases use optimized vector routines.

The results of the low pass filter tests in Table 1 show Gedae-RTL is useful in creating efficient code for feedback loops. Because results from the previous operation are needed to compute the current

operation, this algorithm must run at granularity 1. Three implementations were tested – a subgraph with four boxes (`delay`, `sub`, `add`, and `multK`), the primitive from the Gedae library (`lpf`), and a Gedae-RTL implementation (shown in Figure 1 above). Since no optimized routine exists in the E library for this operation, the `lpf` primitive also uses vanilla C-code; due to the amount of indirection used, the Gedae-RTL implementation is slightly faster. The subgraph implementation is significantly slower due to the overhead of firing four boxes per token processed.

Table 1 - LPF Execution Times on

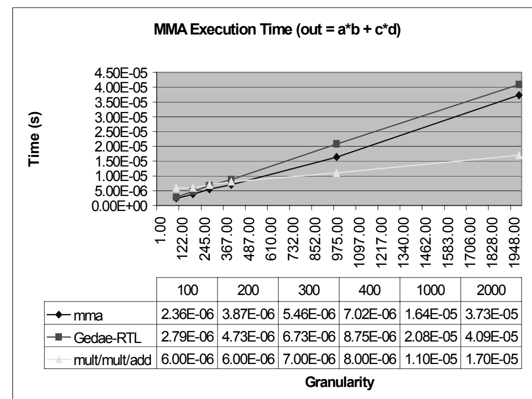


	100	500	1000	2000
lpf	4.01E-06	1.73E-05	3.39E-05	6.73E-05
Gedae-RTL	2.25E-06	8.68E-06	1.65E-05	3.22E-05
sub/add/multK	8.00E-04	4.00E-03	8.00E-03	1.60E-02

The results of the other two tests provide our motivation for exploring target-based optimizations. Both the MMA operation and Sobel operation can be processed at high granularities. The MMA operation has no delays. The delays in the Sobel operation create a feed forward loop which still allows for vector processing. The vector routines needed to do the addition and multiplication necessary for these operations are included in the E library.

In the graph in Table 2, the results of the MMA operation are shown. Again, three implementations are tested: a subgraph with three boxes (2 `mult`'s and an `add`), a primitive (`mma`), and a Gedae-RTL graph. Not only is the Gedae-RTL implementation slower than the `mma` primitive, the slope of its execution time growth is also higher than that of the subgraph implementation.

Table 2 - MMA Execution Times on Mercury-Altivec Hardware



	100	200	300	400	1000	2000
mma	2.36E-06	3.87E-06	5.46E-06	7.02E-06	1.64E-05	3.73E-05
Gedae-RTL	2.79E-06	4.73E-06	6.73E-06	8.75E-06	2.08E-05	4.09E-05
mult/mult/add	6.00E-06	6.00E-06	7.00E-06	8.00E-06	1.10E-05	1.70E-05

Target Optimization

The results in Table 2 demonstrate that Gedae-RTL does not provide an efficient general purpose programming language for DSPs. In the current code generation process, one could create an LSP specific to the AltiVec architecture studied above and improve the results. For example, to program C-code for the AltiVec, an AltiVec LSP could create a for-loop that processes four words at a time, filling the four processing chains, like the following C-loop that implements an optimized vector add:

```
VA = (const vector float *)A;
/*Altivec type: 4 words*/
VB = (const vector float *)B;
VC = (vector float *)C;
for (i=0; i<N; i+=4) {
  *VC = vec_add(*VA, *VB);
/*Altivec library function*/
  VA++; VB++; VC++;
}
```

While the code above is target optimized, there is still a large reliance on the compiler to implement the optimizations. Also, to accomplish this code generation, the LSP must be augmented with a large amount of target specific information.

It is not desirable to add architecture- and vendor-specific optimizations directly to the LSP. Part of Gedae's power is that all its

optimizations (e.g., E library, subscheduling, exclusive families, etc.) are available to all targets. If the LSP contains the information needed to fill the four data paths of an AltiVec, this information is not available for another architecture with a similar structure. Gedae must provide a generalized solution for providing these optimizations instead of keeping this intelligence centralized in the code generation.

For further motivation for moving the optimization from the code generation level to the graph analysis level, let's consider the build process for a C-based LSP. This build process is shown in the figure below. Once the code is created by the LSP, the vendor tools must take many steps to turn the target-optimized code into efficient machine code. The vendor tools shown to the right are based on those used by the Gnu C Compiler (GCC). These vendor tools work like a black box, giving the user little visibility or control to the optimizations taken, and in many ways, the LSP must be tuned to the vendor tools to produce reliable results (as far as performance).

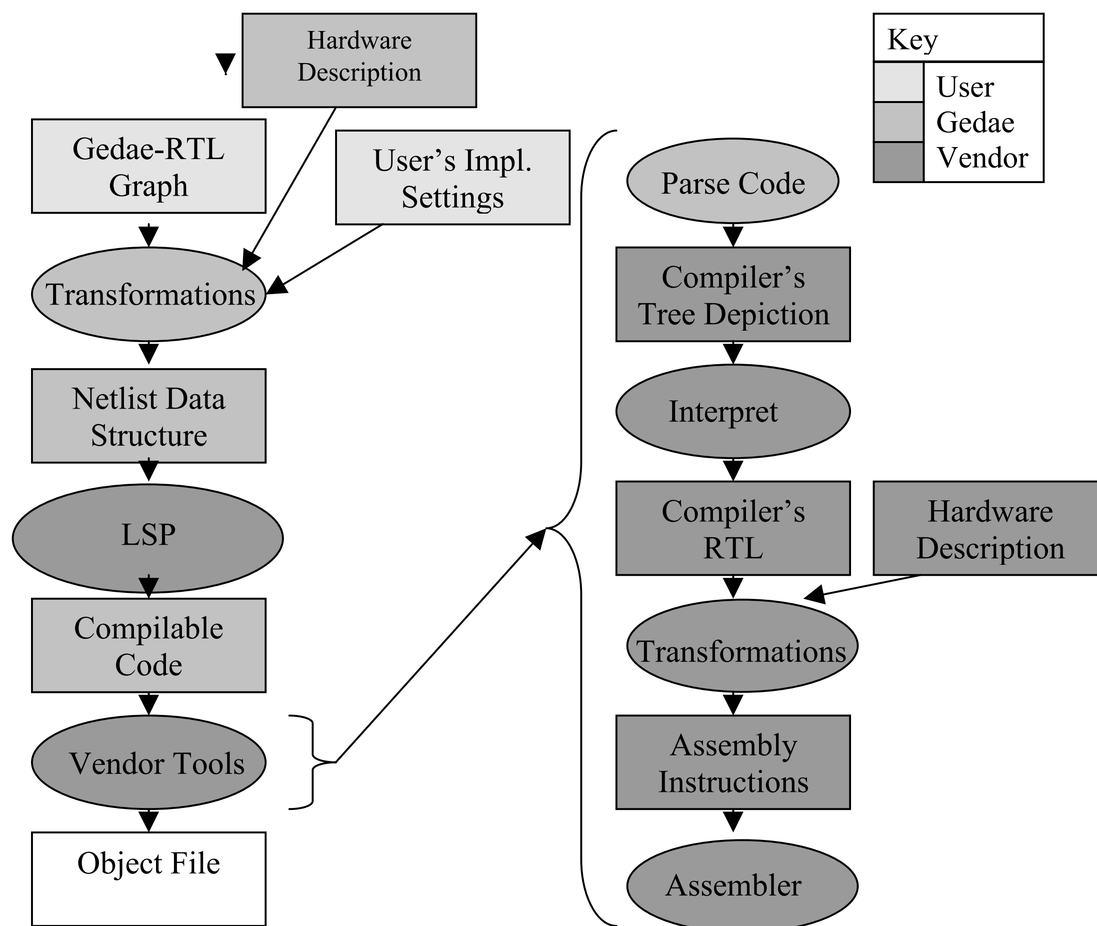


Figure 3 – Code generation and compilation with GCC

As shown in the figure, GCC and many other C-compilers translate C code into an internal RTL depiction. This RTL depiction is transformed according to target specific rules (GCC calls them "passes" of the data structure) in order to create the final assembly instructions. While this

framework will allow target-optimized code to be created and built, many steps are taken to translate Gedae-RTL to C-code only to have the vendor tools translate the code to its own RTL language.

The Gedae-RTL data structure can be transformed according to the hardware just as well as other RTL data structures. Thus, if more intelligence is built into the Gedae-RTL transformations, the Gedae-RTL build process can be streamlined like in the figure below. This streamlined build process provides a simplified means of compiling an algorithm, removing the code interpretation steps necessary in compilers while still providing a high-level means of defining the functionality. Through Gedae, the developer will also have more visibility and control over the optimization and compilation of his code.

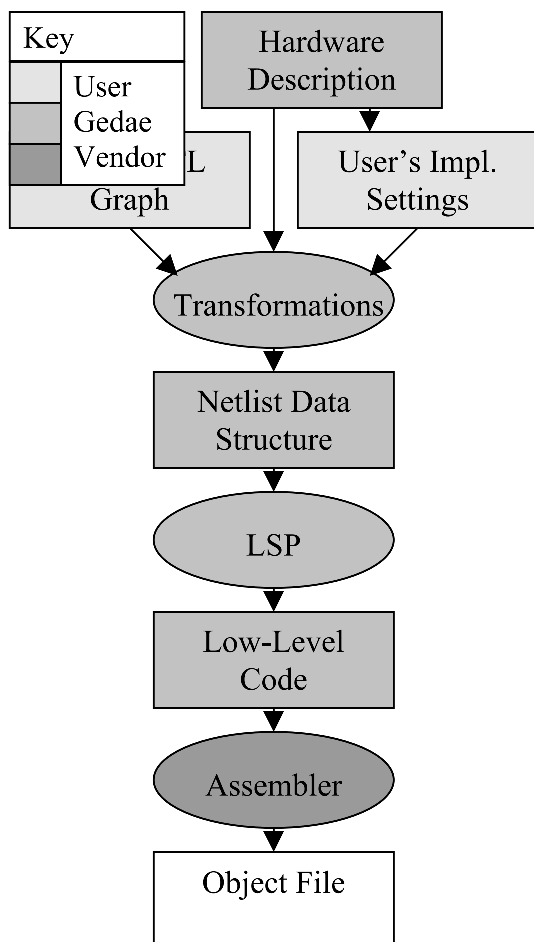


Figure 4 – Code generation and compilation structure with less reliance on vendor tools

Extension of the Hardware Description

Gedae serves as an example of how to produce an architecture independent application. Gedae does this by implementing the application to run on a virtual machine. The virtual machine consists of a Runtime Kernel (RTK), vendor components, and the hardware. The implementation process that Gedae uses is shown in the figure below.

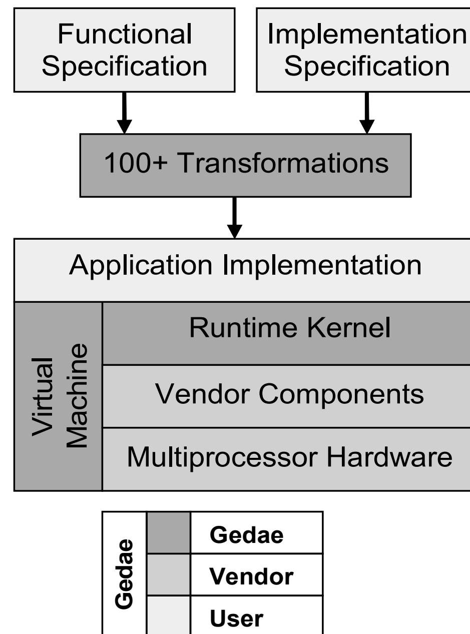


Figure 5 – Core Gedae implementation process

The RTK is a set piece of C-code that is recompiled for each target as part of the Board Support Package Development Kit (BSP Kit). The RTK is run on each target processor and controls when Methods of primitives are executed. A single piece of C-code cannot be efficient on all targets, so in order to provide efficiency, the virtual machine is highly parameterized. For example, if a single transfer method was implemented, it may not work on all targets, and it definitely won't be the most efficient transfer method for all targets. Thus, as part of the BSP Kit, multiple transfer methods can be defined, and the graph developer can choose at development

time what transfer method is used through the Group Control dialog.

The concepts behind the virtual machine can be extended to how we program hardware with Gedae-RTL. To direct these transformations of the Gedae-RTL graph, the "hardware description" must provide Gedae with extensive knowledge of the architecture. The embedded configuration file currently provides a system level view of the architecture such as is shown in the figure below. This view defines each processor and how the processors are connected.

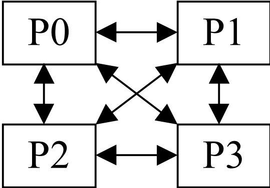


Figure 6 – The system level view shows the processors and their interconnections.

To allow for target-based optimizations based on this information, we must provide Gedae with information on the processor level. One can view a processor as a set of components that nominally includes registers (both data and address), ALUs (which may perform different functions), memories, and connections between these components. We can create a model of a processor that includes those components. The model can be parameterized so that any hardware can be studied by altering the parameters.

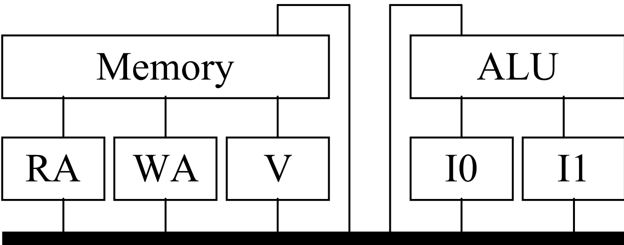


Figure 7 – The processor level view shows registers, memories, and ALUs.

If the embedded configuration file is extended to provide Gedae with knowledge of the architecture down to this level, that information can be used by Gedae-RTL when transforming the functionality into implementation to make target-based optimizations.

Gedae uses this embedded configuration file, in part, to parameterize the virtual machine. With these extensions to the embedded configuration, the virtual machine is now not just a collection of programmable units loaded with a RTK, but each programmable unit is defined as a collection of parts, as in Figure 7.

When Gedae transforms the Gedae-RTL graph into a definition of the implementation, it is creating machine code for the virtual machine. It is creating a long instruction word which defines how each part in the processor is enabled/disabled (what operation the ALU performs, what registers load values from the bus, whether memories are reading or writing, etc.).

To create the program code, the long instruction words (or "virtual machine code") must be translated into the assembly or C routines the assembler or compiler can process. Each long instruction word in the virtual machine code will relate to a set of short instruction words in the actual machine code, so Gedae will do another transformation, translating the abstract long words into a sequential set of compilable short words.

This planning of the control of the ALU, memory, and registers is very analogous to the behaviour of the RTK. The RTK is given a predefined set of functions it can run – the set of primitives in the graph – and it must coordinate the movement of data between queues – both internally and incoming from other processors. Similarly, Gedae-RTL is given a predefined set of functions it can run – the ALU functions

defined by the embedded configuration and used in the primitives – and it must coordinate the movement of data between registers and memories to perform the functions.

Examples of Transformations

The developer's Gedae-RTL graph is transformed an internal netlist data structure which implements the functionality on a virtual processor, as defined and parameterized by the hardware description. These transformations serve to both perform the implementation and optimize it according to the hardware description.

One basic transformation is the inclusion of BSP components. The Gedae-RTL depiction declares input and output streams, parameters, local memory, connections to input and output pins (on FPGAs), and other basic behaviour. These components are declared in the syntax of the graph, but the code that implements them is not provided or included by the developer. Instead, Gedae takes the netlist exported from the editor and augments it with primitives which implement the corresponding BSP components, replacing the declaration with code that implements the desired functionality. Here we see the interaction between the BSP and the LSP. The LSP is not aware of any architecture- or vendor-specific code; instead, Gedae pulls in the appropriate BSP components and the LSP is left to simply output the contents into code.

Another example that shows a transformation necessary for implementation is the serialization of operations. The Gedae-RTL depiction shows a graph of concurrent operations, with registers retaining state and creating the delays that form the pipeline. Part of the transformation from specification to this internal data structure includes a sequential ordering of the operations in the primitives necessary for creating serial code for the target. If we are creating VHDL code for an FPGA, this sequential ordering is ignored, but if we are generating Ansi-C or assembly code, the ordering provided in the internal data structure is necessary for creating working code.

For an example of an efficiency enhancement possible in the transformation of Gedae-RTL graphs, consider the use of intermediate memory in Gedae applications. If the user creates a graph with a vector add following a vector multiply, then there is an intermediate buffer between the add and multiply. If the processor has sufficient ALUs and registers to complete both operations without the use of memory, the efficiency can be greatly improved by implementing the algorithm without the intermediate buffer.

If the entire functionality is implemented in Gedae-RTL, there is newfound opportunity to remove the intermediate buffer. Figure 8 shows an implementation of the vector add/multiply in Gedae-RTL. Stream-to-vector (s_v) and vector-to-stream (v_s) conversions read and write data from the buffers.

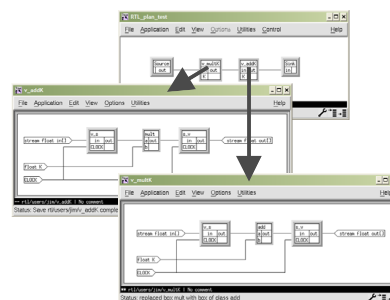


Figure 8 – A vector mult-add operation creates an unnecessary intermediate buffer

The implementation in the figure above would be equivalent to using the existing C-based primitives `v_multK` and `v_addK`. In other words, two for-loops are executed, the first reads from memory, does the multiplication, then writes the result to

memory, and the second reads from memory, does the addition, then writes the final result to memory.

To optimize the graph, Gedae will notice the s_v box follows the v_s box and that the add and multiply can be implemented on the target without the intermediate buffering. A Gedae-RTL graph that represents this optimization is shown in the graph below. The two graphs above have been combined

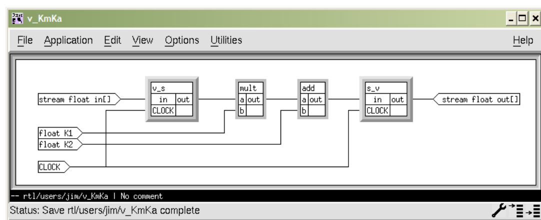


Figure 9 – The Gedae-RTL graph can be transformed to remove unnecessary intermediate buffers

Expand the Scope of Algorithms Programmable by Gedae-RTL

A language capable of being translated into target-optimized code is only valuable if many algorithms can be programmed in the language. Our goal is to reach equivalent expressiveness with Gedae's core primitive language, i.e., any C-based Gedae primitive should be specifiable in Gedae-RTL.

The Gedae-RTL extension allows developers to create flow graphs which process the buffer one sample at a time. Currently, all Gedae-RTL processing uses scalar tokens, which greatly limits the expressiveness of the language. Memory allocations can be defined using `stream` declarations or using the `M`-function. However, only one element can be read from memory at a time.

For example, consider how to program a matrix transpose. Figure 10 shows how we would like to implement the algorithm. A small amount of logic tells the read modules what order to read from the buffers, and enable switches tell the read and write modules when the buffer is read to be read (i.e., full). This functionality cannot currently be directly implemented in Gedae-RTL because there is no notion of token size, and thus no enable switch or capability to read in different orderings.

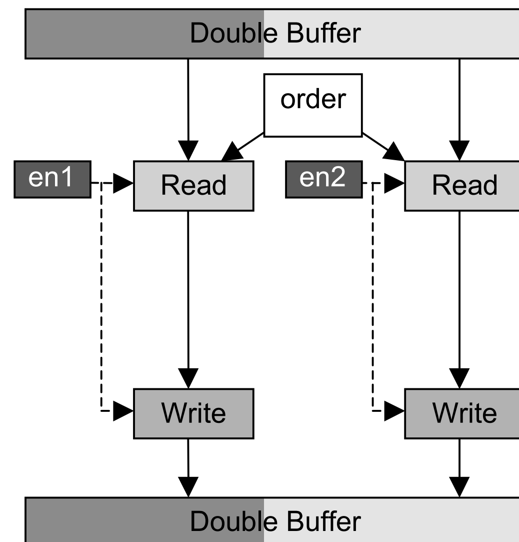


Figure 10 – A matrix transpose cannot currently be implemented in this manner in Gedae-RTL

A second example is a vector sum of elements, as shown in Figure 11. Unlike the matrix transpose, here we can start adding as soon as data arrives. However, there currently is no way to specify the static N:1 relationship between reads and writes in this operation. Gedae-RTL can currently only implement this algorithm with a dynamic output, which reduces the efficiency. Thus, if static interpolation and decimation rates were a part of the Gedae-RTL language, a larger number of algorithms could be programmed with the language.

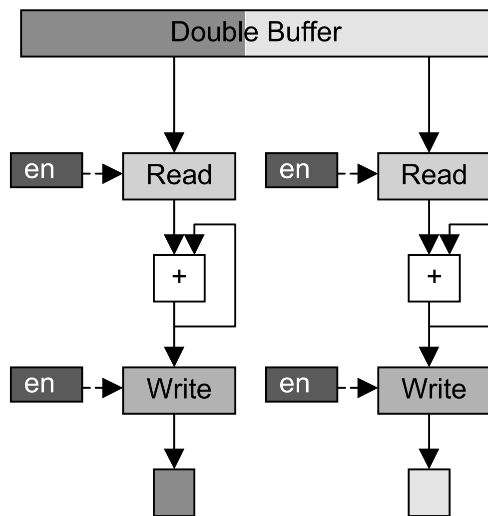


Figure 11 – The N:1 relationship between reads and write can currently only be implemented dynamically in Gedae-RTL

These port characteristics – token type and interpolation/decimation – are two of many such port characteristics that would be useful in Gedae-RTL. Also, the inplace operator states that the output buffer uses the same memory locations as the specified input buffer. As another example, overlap and hold allow the developer to retain input and output tokens (respectively) from previous execution in the buffers.

Future Plans

This paper lays the framework for developing Gedae-RTL, the hardware description, and their integration so that more efficient code can be generated from the language.

First, the core Gedae port characteristics must be integrated into the Gedae-RTL language. A set of components must be created that implement the port characteristics, and transformations must be created to incorporate those components into the user's graph. Once these port characteristics have been integrated into the language, a wider range of algorithms can be implemented in Gedae-RTL.

Next, the hardware description must be implemented. First, we will study an example DSP, such as the Sharc or AltiVec, and investigate its architecture and how algorithms can be most efficiently implemented on it. A set of benchmark algorithms will be created during this investigation to monitor the efficiency gains. A method for mapping the Gedae-RTL graph to the hardware will be created, so that Gedae can determine and optimize the data movement through the registers, memories, and ALUs when transforming the user's specification into an efficient implementation.

References

1. Free Software Foundation, 2004, GCC Manual
2. Jaenicke R, 1999, Programming Applications for the AltiVec Architecture
3. Steed J, 2003, Exploration of Gedae-RTL as an Efficiency Enhancement
4. Steed J, Lundgren W, Barnes K, 2004, Gedae: A Tool For Implementing Software Radio on Heterogeneous Systems, SDR2004
5. Steed J, Lundgren W, Barnes K, 2005, Gedae Hardware Programming Capability, GUC II
6. Steed J, Lundgren W, 2004, Proposal for Gedae-RTL Development

Acknowledgements

The work reported in this paper was funded by the Electro-Magnetic Remote Sensing (EMRS) Defence Technology Centre, established by the UK Ministry of Defence and run by a consortium of SELEX Sensors and Airborne Systems, Thales Defence, Roke Manor Research and Filtronic.