Combining P2 and RDL to build Dataflow Hardware Programs

Greg Gibeling, Nathan Burkhart and Andrew Schultz
University of California, Berkeley
{gdgib, burkhart, alschult}@berkeley.edu

1. Introduction

In this paper we describe our re-implementation of the P2 [1] system and the Overlog declarative networking language on top of the RAMP Description Language [2] (RDL), which can be compiled to a hardware implementation.

Nearly all sufficiently large hardware systems, such as those RDL was designed to support, are built on the globally asynchronous, locally synchronous design pattern because it allows components of the system to be constructed and tested independently. Recently, projects like Click [3, 4] and P2 [1, 5-7], have explored the construction of traditionally monolithic software systems using dataflow components, with a similar communications pattern.

What's more these software systems have admitted a certain performance penalty for the ease of specification and debugging that a dataflow execution model provides. In order to recapture this lost performance, expand the range of applications for these systems and improve the networking functionality available to reconfigurable systems programmers, we have built a compiler which will transform a P2 Overlog specification into a high-performance hardware implementation.

Click was targeted to building router control planes and P2 to build overlay networks (e.g. Chord [8], Narada Mesh, etc) in a succinct and analyzable fashion.

RDL was designed to support large scale multiprocessor computer architecture research, allowing independent researchers to build and assemble complete accurate hardware systems, rather than resorting to system simulation, which is typically several orders of magnitude too slow for applications developement.

Systems like P2 and Click add value by expressing the system as a composition of simple elements executed as a dataflow eases design and implementation at the cost of overhead. Additionally, the parallelism in the dataflow model is difficult to manage in a microprocessor [9]. This project takes the logical extension of expressing the high parallelism inherent in dataflow models directly in a parallel medium, namely gate level hardware. We show that it is possible to automatically implement complex systems in hardware and obtain a substantial performance benefit by harnessing the implicit parallelism of these systems.

2. Background

This project represents the synthesis of several areas of research, namely distributed systems, languages, databases and computer architecture. This section provides background on the various projects which form the basis of our work.

In this paper, we present an alternative implementation of the Overlog language and semantics which can be compiled through RDL to Verilog for implementation on an FPGA. Implementing overlay networks in hardware has two direct benefits. Because the hardware implementation is specialized and parallel, it can run orders of magnitude faster than a comparable software system. Second, a hardware overlay network would provide a key component of large scale reconfigurable computing clusters such as the BEE2 [10] used by the RAMP project [2, 11].

2.1 P2: Declarative Overlay Networks

In the past several years, research in overlay networks has changed the way distributed systems are designed and implemented. Overlay networks provide many advantages over traditional static networks, in that they enable highly distributed, loosely coupled operation in a robust, conceptually simple manner [1, 8, 12, 13]. However, despite the conceptual clarity that overlays provide their implementation is typically a complex and error prone process.

P2 and Overlog were designed specifically to solve this problem. P2 uses a high level language, called Overlog, to specify the overlay network protocol in a declarative fashion. P2 essentially separates the description of the overlay from its implementation, making it easier to reason about the correctness of the protocol. Furthermore, P2 automates the implementation of the overlay by compiling the declarative description into a dataflow execution. Other projects such as Click have shown the value of dataflow execution models for simplifying the construction of complex systems.

Aside from the complexity problems, overlay networks typically have performance issues and high implementation costs. Because these networks often maintain a large amount of state and a different routing topology on top of the already costly TCP and IP protocols, they tend to have low performance. Additionally, the generality offered by a dataflow model comes with performance costs, especially when serialized

to run on a microprocessor, thereby losing most or all of the parallelism.

In order to integrate with the current hot topic applications like firewalls, 10Gbps routers and intrusion detection systems higher performance implementations of overlay networks are required. Worse, the complexity and cost of these implementations often forces constraints on the size of the test bed which can be constructed thereby limiting the reliability of the protocol..

2.2 RAMP: Research Accelerator for Multiprocessors

The RAMP [11] project is developing the infrastructure to support high-speed emulation of large scale, highly parallel systems. The RAMP Design Framework is structured around loosely coupled units, implemented in a variety of technologies, communicating with latency insensitive protocols over well-defined channels. In this section, we describe the goals and implementation of the RAMP Design Framework (RDF) as embodied by the RAMP Description Language (RDL) and its compiler, both of which are integral pieces of this project as well as RAMP.

The primary motivation for RAMP [14] is to replace software based architectural simulations, which are 3-5 orders of magnitude slower than ASICs, which are too expensive to use in development of massively parallel multiprocessor systems. This will allow operating systems and applications researchers to work with new architectures before a full system can be built, and allow computer architects to reassess long held assumptions in the face extreme parallelism.

In order to support the RAMP project goals, the framework must support cycle-accurate emulation of detailed ("real hardware"), parameterized machine models and rapid functional-only emulations. The framework should also hide changes in the underlying implementation from the designer, to allow groups with different hardware and software configurations to share designs, reuse components and validate experimental results. Finally, the framework should not dictate the implementation language chosen by developers.

The solution for RAMP was to develop a decoupled machine model and design discipline, together with an accompanying RAMP Description Language (RDL) and compiler (RDLC) to automate the difficult task of providing cycle-accurate emulation of distributed cross-platform communicating units.

2.3 RDL

A RAMP target design is structured as a series of loosely coupled units which communicate using latency-insensitive protocols implemented by sending messages over well-defined channels. Figure 1 gives a simple schematic example of two such units communicating over a channel.

In a typical RAMP design, a unit will be a relatively large component, consisting of about 10,000 gates, e.g. a processor with L1 cache, a DRAM controller or a network interface, basically any subset of hardware that requires tight coupling or a dependence on specific timing. However our experience with RDL has been that it is equally useful for smaller units, such as the tuple handling elements documented in section 5.

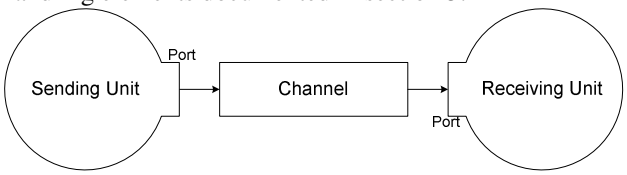

Figure 2: Simple RDL Model

In RDL all communication is via messages sent over unidirectional, point-to-point channels, where each channel is buffered to allow units to execute decoupled from each other. In a design with small units, like ours, the buffering inherent in the channel model forces delays and increased circuit size. However given the relatively abundance of registers to LUTs in most FPGAs, such as the Xilinx Virtex2Pro, the buffering is not a problem, and the increased latency is less important because the P2 model admits pipelining of operations on tuples.

Given such a regimented model of communication, computation and timing, the RDL Compiler can automatically build the complete communications network even between units implemented in hardware and software as shown in figure 2 below. In addition debugging tools will be able to monitor and inject data on channels, by virtue of their well known semantics and opaque implementation.

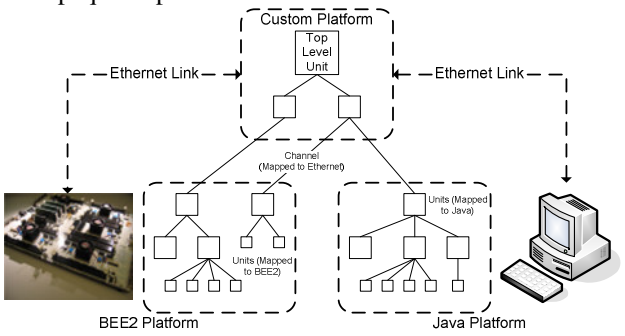

Figure 2: Cross Platform RDL

As shown in Figure 2, RDL makes a clear distinction between the design being emulated or simulated, the "target" system, and the platform which is performing the emulation, the "host" system. In this paper we will restrict our discussion to FPGA host implementations of Overlog targets. In fact we also spent some time on the code necessary to produce Java host implementations of Overlog targets, but the java output functionality was temporarily removed from the RDL Compiler during a major revision to support this project.

2.4 BEE2

The BEE2 [10, 15] is the second generation of the BEE FPGA board originally designed to support in-circuit emulation of radio controllers at the Berkeley Wireless Research Center. The BEE2 was designed to support general purpose supercomputing and DSP applications in addition to the specialized ICE functionality of the BEE.

The BEE2 is also the primary board to be used in the RAMP project. With 5 Xilinx Virtex2Pro 70 FPGAs, each with two PPC405 cores, up to 4GB of DDR2 DRAM and four 10Gbps off-board Infiniband or 10Gbps Ethernet connections, the board includes over 180Gbps off board bandwidth, and 40GB of RAM, enough for even the most demanding applications. Furthermore the bandwidth on and off the board has been carefully balanced to avoid the bottlenecks which often plague such systems.

Because the BEE2 is aimed to be primary RAMP host platform, we have used it as our test platform. Given the speed and implementation density of Overlog designs relative to the BEE2's capacity, it might also provide a useful test platform for overlay networks (see section 3.1).

3. Applications

In the previous section we outlined the various projects which are key components of our work. In this section we expand on this to suggest the ways in which our work will contribute back to these projects.

3.1 Overlay Networks

Because a parallel hardware implementation of an Overlog program can run orders of magnitude faster than the original software implementation of P2, our works opens up the possibility of running experiments on Overlog programs in fast-time. Furthermore, since a hardware implementation does not time-multiplex a single general processor, more nodes can be packed onto a BEE2 board than can be run on a normal CPU. Time and space compression could allow testing of larger networks than current clusters of CPUs can offer.

In addition, fine grained (hardware clock cycle level) determinism, which is a core part of the RDL model, would allow cycle accurate repetition of tests, a great boon to those debugging and measuring a large distributed system.

Line speed devices like routers, switches, firewalls and VPN endpoints could benefit significantly from the parallelism and speed of these implementations combined with the high level protocol abstraction provided by Overlog. For example, this could allow the design of core-router protocols using a simple declarative language, and the automatic generation of 1-10Gbps, line rate, implementations of these protocols.

3.2 Distributed Debugging Tools

In [16] some of the original P2 authors present a debugging framework for Overlog designs which makes use of reflection to debug Overlog designs using Overlog and the P2 infrastructure. Of course this should be a natural idea given the ease with which such a declarative specification captures the semantics of distributed systems, exactly like the way debugging checks need to be specified. While the reflection and tap architecture presented in [16] is unsuitable for implementation in hardware, we believe that similar concepts will be appropriate for debugging general RDL designs.

The reflected architecture is unsuitable for general RDL first because the meta-information even for a single hardware node could quickly overwhelm the storage available at that node, both in capacity and bandwidth. Even invoking the RDL capability to slow target system time, this would produce generally poor performance. Second, the ability to add and remove dataflow taps which is so simple in software is prohibitively complex, even in reconfigurable hardware[1]. In addition, to support code reuse, RDL designs admit arbitrary hardware units, including unknown state. This would prevent tracing as presented in [16], as the cause and effect relationships between messages is unknown.

However, even with these limitations the RDL model can easily support interposition on channels for monitoring or data injection. Overlog, or a similar language, with support for RDL message types, could provide a concise and understandable mechanism for specifying watch expression, logging and breakpoints with complex distributed triggers. In this case a hardware implementation is a necessity not only for interfacing with the circuit under test, but also for maintaining the data rate which will often well exceed 10Gbps.

3.3 Computing Clusters

The major drawback of reconfigurable computing platforms, the BEE2 included, has and continues to be the infrastructure required to perform computation on these boards. The memory and network remain the two main peripherals to FPGAs, and the two hardest pieces of hardware to interface to. This project aims to alleviate the situation for networking, by bringing a higher level of abstraction, namely Overlog, to bear on the problem.

RDL and RDLC obviate a large portion of the communications complexity by providing a uniform channel abstraction over a variety of implementations. However, the point-to-point model of communication in RDL cannot support dynamic topologies.

Simplified protocols force the use of highly controlled networks to avoid packet loss or corruption, which these protocols cannot cope with. In a 1000 node RAMP system this kind of restriction would be prohibitive. Providing hardware implementations of high level overlay networks could allow their use for general

[1] This is a side effect of commercial FPGAs and a lack of applications in this area, not a fundamental limitation of the technology.

communications, replacing the fragile, unreliable static protocols normally used with robust, adaptable overlays.

4. Languages & Compilers

In the previous sections we presented the enabling research and motivating applications for our work. In this section we switch to a more concrete discussion of the code base, including both of the main compilers used in this project.

4.1 RDL and RDLC2

In section 2.3 we gave a rough outline of the semantics and model for the RAMP Description Language, RDL. This section documents the second major revision of the RDL Compiler (RDLC2) which produces Verilog implementations from RDL source.

Because one of the goals of the RAMP Design Framework is to tie together existing designs, RDL is not a behavioral language, it only includes constructs for instantiating and connecting units in a hierarchy. The implementations of units at the leaves must be written in a RDL host language; Verilog in this paper.

Because RDLC2 is meant to translate RDL into almost any other language (Verilog, VHDL, Java, C, C++, BlueSpec, etc) it is structured as a general compiler framework, with support for the chaining of code transformations, I/O in multiple languages and a robust plugin architecture.

At the core RDLC2 provides two commands: *shell* and *map*. *Shell* takes a leaf unit specification and produces the Verilog shell into which the unit's implementation must be written. *Map* takes a complete system specification and generates an implementation of it for the specified platform.

Shown below is the simplest RDL example: a counter. In this design, a unit representing a button and switch is connected to a counter unit, which is in turn connected to a display unit.

```
unit <width> {
    instance IO::BooleanInput
        BooleanInputX(Value(InChannel));
    instance Counter<$width>
        CounterX(InChannel, OutChannel);
    instance IO::DisplayNum<$width> DisplayNumX;
    channel InChannel;
    channel OutChannel { -> DisplayNumX.Value };
} CounterExample;
unit <width = 32, saturate = 1> {
    input bit<1> UpDown;
    output bit<$width> Count;
} Counter;
unit {
    output bit<1> Value;
} BooleanInput;
unit <width = 32> {
    input bit<$width> Value;
} DisplayNum;
```

Notice that units can include parameters, in this example the width and saturate parameters to the counter control the bit width and whether it saturates at 0. Not shown in this example are external connections, which allow the BooleanInput and DisplayNum units to connect to chip level wires and thus perform user visible I/O.

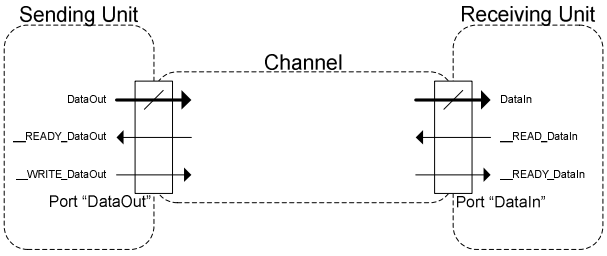

Figure 3: Shells

Shown in figure 3 above are two unit shells connected by a channel. The key point of this diagram is the control signals for the channel, which exactly match the synchronous FIFO semantics of the channel-unit interaction. This simplicity of this interface, and the temporal disconnection of the units at either end of the channel are key to the ease with which RDL units can be implemented, and a big part of the success of this project.

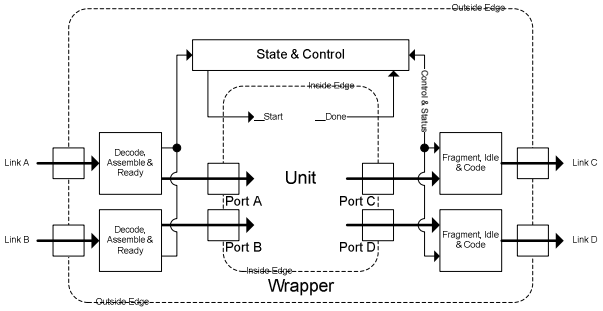

Figure 4: Unit Wrapper

In addition to shells, which are generated prior to unit implementation, the other output of RDLC2 are the unit wrappers, which are generated by the map command. Wrappers include the logic to marshal and unmarshal the structured (struct, union and array) RDL messages types from simple fixed width bit vectors. Wrappers also include the start and done logic, which is responsible for maintaining the timing model required to keep the target system deterministic in the face of host level communications uncertainties.

The map command may also invoke a series of plugins designed to implement specialized units. This functionality is used to generate e.g. small SRAMs and FIFOs, with uniform semantics but platform specific implementations. This is also used to generate some of the more complex Overlog elements documented in section 5. In truth this borders on allowing behavioral

specifications in RDL, however these plugins must still be specialized to each output language family[2].

In many respects RDL is similar to the Liberty Structural Specification Language [17] or the Click [3] language. The differences are in the timing model and high level data types provided by RDL.

4.2 Overlog

Overlog is a variant of Datalog designed to manipulate database tuples, implementing distributed inference over a set of relations. An Overlog program consists of a set of relation declarations, where each relation is a materialized table or a stream of tuples, combined with a set of inference rules of the form:

```
Name Relation1@N(A, B + 1) :- Relation2@N(A, B);
```

This rule specifies that a tuple being added to relation Relation2 at node N should result in a tuple being added to Relation1 at node N, with the relevant fields. Notice that both relations 1 and 2 could be materialized tables or tuple streams.

In the original Overlog syntax given in [1], only materialized relations need to be declared and even then they are un-typed. Firstly, because we are generating hardware, which should be efficient, we require that the types of relation fields be declared ahead of time. Secondly, in order to simplify the planner, and catch a larger portion of errors at compile time, we required tuple streams to be similarly declared. Examples of materialized table and tuple stream declarations for our modified dialect of Overlog are shown below.

```
materialize TName [10] for 10 (key Int, Int);
stream SName (Int, Bool, NetAddress);
```

While most of the hardware implemented at the time of this writing can handle un-typed tuples more interesting features like paging materialized table storage out to DDR2 SDRAM to support very large tables would be costly without a certain minimum of type information.

More importantly, in the short term these declarations have allowed us to catch a number of mindless typos and programmer errors at compile type. The dangers of poor type checking in hardware languages are all too real, as Verilog provides almost non-existent and non-standard type checking.

As a final exercise we present the Overlog program fragment shown below, which is an extension of one of our tests. It declares two streams, tells the compiler to put a watch of each of them for debugging during simulation, specifies some base facts, and a simple rule for computation.

```
stream Stream0(Int, Int);
stream Stream1(Bool);

watch Stream0;
watch Stream1;

Stream1(true);
Stream0(0, 1);
Stream0(2, 3);
Stream1(false);

Stream1@N(A > B) :- Stream0@N(A, B);
```

The expected output of this program is shown below. However the interleaving of the results will differ based on the actual execution timing.

```
Stream1: <true>
Stream0: <0, 1>
Stream1: <false>
Stream0: <2, 3>
Stream1: <false>
Stream1: <false>
```

In section 5, we discuss the details of our Overlog planner, and the architecture of the resulting system, but we must also briefly touch on the integration of the Overlog and RDL compilers. In section 4.1, we described RDLC as a compiler framework with support for a plugins. These features allow us to specify the Overlog compiler as a chain of program transformations turning an Overlog program into and RDL design which is then turned into a Verilog design. In addition, this chaining could easily support Overlog rule rewriting such as the localization described in [6]. Finally the plugin architecture allows us to specify the static portions of the RDL design using RDL which includes plugin invocations to fill in the implementations based on the Overlog.

This pattern of RDLC2 transformation chains and plugins has also been used to implement a computer architecture compiler with an integrated assembler. We believe it is general enough to support nearly any transformation required.

The Overlog compiler contains four distinct components which are chained together. First, the front end lexes and parses the inpu. Second a resolve transformation performs error checking and variable dereferences. Third, the Overlog planner plugin is invoked by the RDL portion of the compiler to fill the top level P2 unit in with the various units required implement the Overlog program. And finally, a series of hardware generator plugins create the actual Verilog unit implementations of the dataflow elements used by the planner.

5. System Architecture

[2] An example family is the hardware family including Verilog and VHDL, which are very similar.

An RDL design is composed of communication units, the lowest level of which are implemented in a host language, in this case Verilog. A P2 system however is composed from a fixed set of elements, assembled to match the input Overlog program.

This section describes the elements we have implemented as RDL units, and the planner transformation which assembles these elements. Shown in figure 5 below is a complete system including the network, several rule strands, a table and the table of base facts which are used to initialize the Overlog program.

As in P2, our implementation consists of a series of linear dataflow sub-graphs called, roughly one per Overlog rule. Each strand starts with a triggering event, and ends with a resulting action. Events include updates to tables, reception from the network or timers specified using the special *periodic* relation.

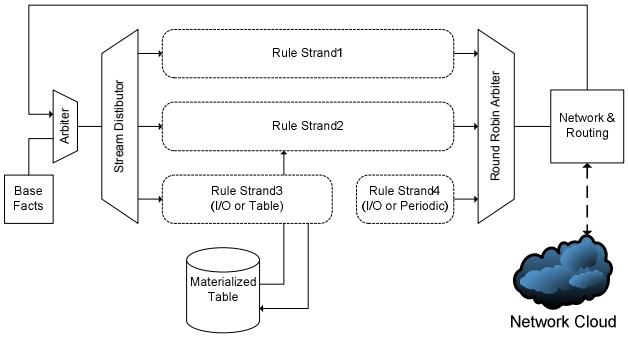

Figure 5: A Complete Overlog Node

5.1 Data Representation

This system includes two programming languages and three abstractions of computation, each with its own data model. At the Overlog level, data is presented as materialized relations and tuple streams which are manipulated with the standard relational operators. We write these tuples as follows.

```
<10.1.1.1, 0, true>
```

This is an example of a tuple with a destination network address, followed by an integer and a Boolean field. Notice that this abstraction omits the details of mapping these tuples to actual wires.

RDL units handle tuples as streams of fields, annotated with type, start of tuple and end of tuple signals. The RDL declaration for these messages is shown below.

```
message <IWidth, TIDWidth, NAWidth> mstruct {
        Data<$IWidth, $TIDWidth, $NAWidth> Data;
        Marker Marker;
} Field;
message <IWidth, TIDWidth, NAWidth> munion {
        bit<$IWidth> Integer;
        ::1::NetAddress<$NAWidth> NetAddress;
        bit<1> Boolean;
        event Null;
} Data;
message mstruct {
        bit<1> Start, End;
} Marker;
```

From this specification of RDL messages, RDLC automatically generates wires of the specified bit widths, for each field of a union or struct. Union fields are muxed down to a single set of wires for transmission and storage in the channel. In order to support this marshaling, RDLC adds a of tag wires and constants which allow a unit or channel to specify which subtype a union currently holds.

Our representation of tuples was designed with two constraints in mind. First, because the original P2 system uses seconds as the time units, time multiplexing hardware is a profitable way to reduce implementation costs without affecting the functionality of Overlog programs. Even with time units in milli-or micro-seconds, bottlenecks due to the serialization of fields are unlikely, given that most modern FPGA implementations run in excess of 50MHz without difficulty.

The second constraint is on the handling of variable length and un-typed tuples. In addition to components like the network and arbiters, which must handle tuples from widely different relations, our early experiences trying to build distributed databases in Overlog suggested that the ability to store dynamically typed tuples would be a valuable feature. By supporting this kind of processing we can allow future work to build run time programmable tuple processing elements. These elements will be costlier due to lack of typing information, hence the addition of types to Overlog in order to reduce these costs where possible.

As a final note, because the bit widths of the values in our system have been parameterized, it is possible to build smaller or larger systems as needed on a protocol to protocol basis, simply by changing the width of integers or network addresses. In the future we believe a more direct translation between Overlog and RDL types would be helpful both for implementation efficiency and for supporting Overlog as a debugging tool for RDL.

5.2 Tables & Storage

Because Overlog is primarily targeted to building overlay networks, materialized tables are slightly different than standard SQL-style tables. In addition to size limits and keys for tuple identity, Overlog includes expiration times for stored tuples.

Providing support for all three of these features in hardware was one of the primary sources of implementation complexity for this project. With the high implementation costs of hash-based indexing structures, and the relatively small size of tuples and tables, we chose to implement all table operations as linear scans.

Shown in figure 6 is the composite table unit, which supports a single input and output. Input requests are represented as an RDL message union of events, which reduces to a set of tag wires with no data.

```
message munion {
       event Scan, Insert, Delete;
} TableRequest;
```

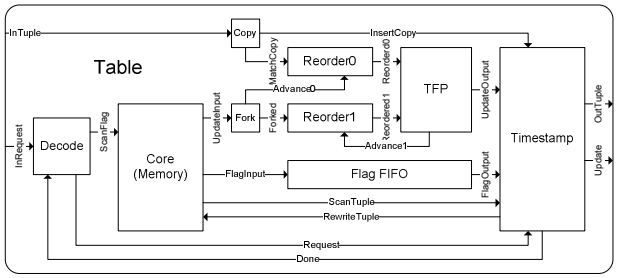

Figure 6: Table Implementation

A scan operation simply iterates over all tuple fields in the table sending them to the output in order. An insert and delete must also perform a join against the input tuple to match keys. In both cases a match implies that the existing tuple should be dropped. Because these operations are implemented as scans, they can in fact perform garbage collection on unused areas of the table memory, by simply rewriting the entire table. Tables can include tuple expiration times, forcing a rewrite on a scan as well in order to avoid outputting a stale tuple.

Because new tuples are always inserted at the end of the table, and the table is garbage collected on each scan, it was a simple matter to implement the drop-oldest semantics for full tables: the oldest tuple will always be the first one in scan order.

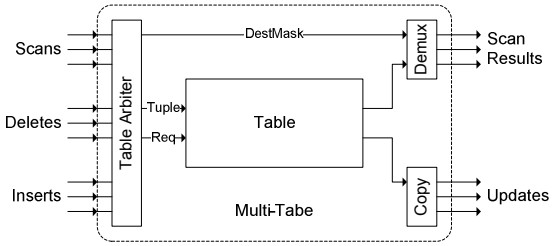

Figure 7: Multi-Port Table Adapater

Shown in figure 7 is the unit used to provide multiple ports to the table, in order to support multiple rules which access a single table. The input stage consists of a round-robin arbiter, modified to allow multiple table scans to proceed in parallel. Since it is common for many rules to use a table in their predicate, this is an important performance optimization.

In our original design system, we intended to pack tuples down to the minimum number of bits, shifting and filling where needed based on types. However, providing support for un-typed tuples made this an expensive proposition, as the implementation would require either a barrel shifter (very expensive in FPGAs) or possibly many cycles per field. Instead each tuple field is stored at a separate address in the table memory.

5.3 Rule Strands

Figure 8 shows a complete rule strand, including the logic for triggering table scans on the arrival of an event, and the tuple operation unit, shown in detail in figure 9.

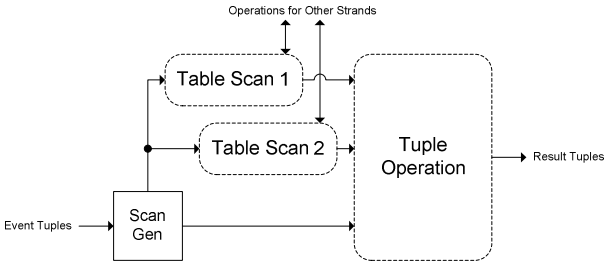

Figure 8: Rule Strand

Because each Overlog rule has at most a single event predicate which is joined with many materialized tables, each strand consists of a series of nested scans which are triggered by the arrival of the relevant event. In Overlog an event can be an update to a table, the arrival of a tuple from another node, or a periodic timer event.

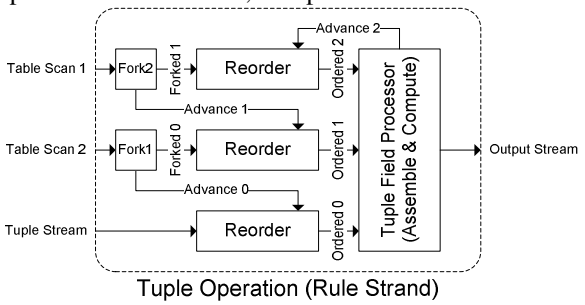

Figure 9: Tuple Operation Unit

Tuples, including the event tuple, and those resulting from scans, are fed into a Tuple Operation unit, which consists of a series of field reordering buffers, chained to implement a nested loops join, and a Tuple Field Processor, which will perform the actual calculations. We describe the Tuple Field Processor in section 5.4.

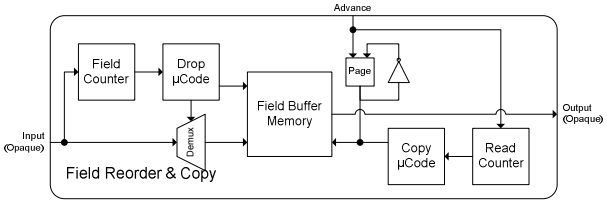

Figure 10: Field Reorder Buffer

Figure 10 shows the implementation details of the field reorder buffer. These buffers duplicate and drop

fields as required to support the calculations and joins specified by the Overlog. For example the rule Result@N() :- periodic@N(A, 10) does not use any of the input fields of the periodic event stream, in which case the reorder buffer for the periodic stream will simply drop all fields. In the rule Result@N(A, A) :- Predicate@N(A, B, C), the field reorder buffer would duplicate the A field, and drop the B and C fields.

The reorder buffers decouple the sequencing of data, which is implied by the output field order, from the operations performed in the Tuple Field Processor. The alternative is direct implementation of the dataflow graph extracted from each rule, with a channel for each variable. However, because tuples will not arrive very close together such a direct implementation would be severely wasteful in FPGA resources to no appreciable benefit.

5.4 Tuple Field Processor

In order to time share the hardware which performs the computation for each Overlog rule, we built a small stack processor generator. While there must still be one such processor per Overlog rule, this decreases the implementation cost by a factor between 5 and 20, depending on the complexity of the rule.

A tuple field processor has three memories: stack, constant table and instruction ROM. It implements relational join, select, aggregate and computations. Projection is handled in the reorder buffers.

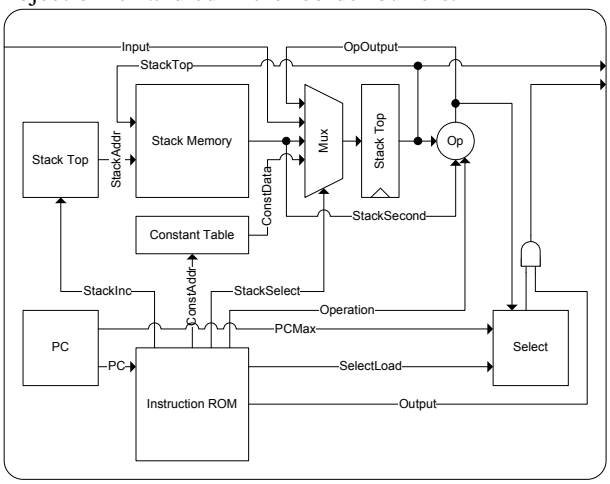

Figure 11: Tuple Field Processor

Figure 11 shows a simplified schematic of a tuple field processor. The operations bubble is specialized for the operations required by the Overlog rule the TFP implements. Furthermore, there is no support for jump, conditional or loop constructs. Without conditionals, selection is implemented by optionally writing output fields based on prior binary selection conditions that result from both joins and Overlog selection clauses.

Using a processor introduces the possibility of loading new transformations in at run time, by adding a port to write incoming tuples to the instruction memory. This would allow an Overlog node to be dynamically reprogrammed without re-running the compiler tools. This is less general than full FPGA reconfiguration, since it cannot change the overall dataflow of tuples, however that could be implemented using mux units.

A significant portion of the Overlog compiler code is actually the compiler from Overlog to a custom assembly language for the TFP, and the code which then assembles and links these programs and builds the specialized TFPs to execute them.

Conceptually, the TFP and its assembly language are very similar to the PEL transforms which are embedded in the original P2 system for much the same purpose. However where PEL transform significantly ease the implementation of Overlog in software, the TFP is an efficiency optimization which reduces the size of the generated circuits by almost an order of magnitude for more complex rules.

5.5 Network Interfacing

The extremely large capacity of modern FPGAs such as the Xilinx Virtex2Pro 70 on the BEE2, enables us to pack many Overlog nodes on each FPGA. This implies that the network infrastructure must span both on-chip and off-chip connections. To this end, we developed a high-bandwidth cross-bar packet switch to connect nodes regardless of their location. As with the components in the nodes themselves, the cross-bar and the network interfaces were designed and implemented in RDL.

Figure 12 shows a simple schematic of the network interface which connects each node to the network and test infrastructure.

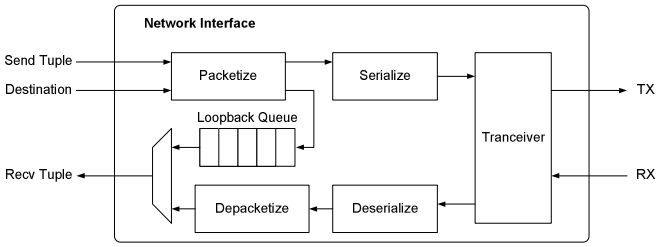

Figure 12: Network Interface

This interface sends packets composed of individual tuples encapsulated inside a tuple containing source and destination information. The tuples come into and out of the interface as a series of fields and are serialized down to an fixed size for transmission through the network.

The switch provides a parameterized number of ports, all fully connected through a cross-bar connection. The choice of a cross-bar enables the highest performance at the cost of increased resource utilization. We have also implemented a “horn and funnel” type switch which uses fewer resources and has lower performance. One of the

key points of this network is that the bandwidth is essentially the network width times the number of crossbar ports, lending itself to creating large, high bandwidth networks very easily.

Overlog nodes are not the only endpoints on this network. Transceivers and "proxies" can be attached to any port, allowing nodes to communicate from FPGA to FPGA on the same board and on other boards. For our test harness, we use this capability to connect the nodes to a Linux system running on the BEE2 control FPGA in order to inject and collect tuples through C programs..

Finally, the switch instantiates a module which specifies a routing policy for each port. For our test system we use a simple policy that routes based on switch port number with a designated default port. More complicated routing policies such as longest prefix match are also possible.

6. Testing

Since our implementation is still in the relatively early phases, we have only run a series of small and synthetic test programs through it. Original we had hoped to run a Chord ring, but the complete Overlog semantics remain more complicated than could be implemented with RDLC2 within a reasonable timeframe (See section 9.3).

6.1 Test Overlog Programs

Our tests consist of several example Overlog programs designed to exercise the Overlog compiler, planner, TFP generator and Table implementation. Of course these tests also cover the vast majority of the hardware unit implementations.

- Facts: This test was designed simply to display Overlog base facts without processing. This is the bare minimum Overlog program, though it does test portions of the networking hardware, and the majority of the infrastructure code. In addition to a sanity check, this provides absolute minimum implementation costs.
- Simple: The simple test consists of a single rule, which fires periodically and increments the sequence number generated by a timer. In addition to the hardware in the Facts program, this includes a periodic timer, and a single Tuple Operation unit, with a single reorder buffer.
- Multi-Stream: Building on the simple test, this program generates a tuple stream by performing some simple calculations on a periodic tuple, and then runs these tuples through two more rules. This primary motivation for this test was to provide latency and circuit size measurements.
- Table: A simple table test, which performs inserts and limited scans over a table which stores a single tuple. This test exhibits a base implementation cost of the table, for size and performance comparisons.
- Join: This test adds a larger table, 20 tuples, and performs a join over these to lookup tuples inserted during a specified time range according to sequence numbers from the periodic source.
- Aggregate: Performs a series of simple aggregates over a table, including count, min and max.
- SimpleNet: A relatively simple network test, which accepts tuples, performs a calculation on them and sends the resulting tuple back to a given address. This was used to test the network interfaces and Linux based debugging tools.

6.2 Test Platform

We are currently using the BEE2 as our test platform, primarily because it is used by the RAMP project. The topology of the BEE2 is such that one of the five FPGAs is designated the "control" FPGA while the remaining four are designated the "user" FPGAs. The control FPGA boots full Debian GNU/Linux on one of the embedded PowerPC 405s in the Virtex2Pro. From this a user can log in to the board, program the user FPGAs and interact with them over high-speed parallel I/Os links.

We have reused infrastructure we originally developed for the RAMP project to connect the tuple network directly to software accessible FIFOs on the control FPGA. The linux kernel on the has drivers which abstract the these FIFOs as either character devices or virtual Ethernet channels. We use the FIFOs as character devices allowing us to write simple C code to inject and read back tuples by reading and writing files. Figure 13 gives a schematic view of this connectivity.

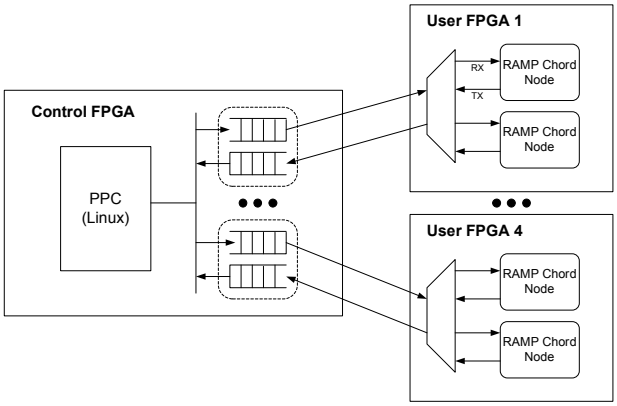

Figure 13: Network Topology

In addition to sending and receiving tuples through this interface, we use the hardware FIFOs to collect statistics and events directly from the hardware. By sharing the infrastructure originally developed for the RAMP project, we have significantly eased the integration of standard Linux software and raw hardware.

Future development with Overlog and RDL will make I/O a first class primitive which can be more easily defined within the language itself. Having I/O defined explicitly in the system description will enable more robust debugging and potentially higher performance communication. By integrating this with the cross-platform capabilites of RDLC, we can also ensure that future projects will be able to share similar communications infrastructure even more easily than we were able to.

7. Performance Results

This section presents performance numbers both on the compiler and the hardware that it generates. Given that this project is still in the relatively early stages, these should be considered rough numbers.

7.1 Compiler Performance

Shown in table 1 below are various compiler and simulation performance metrics for the test programs described in section 6.1. Most of these metrics are tied to the high level system design and the conceptual mapping of Overlog onto RDL. Even without a current basis for comparison, these numbers are important as a baseline for our future work.

Test	Memory	Comp. Time	Load Time	Sim Time
Facts	10MB	5.51s	1:07m	1.68s
Simple	13MB	5.65s	1:15m	2.84s
MultiStream	13MB	15.22s	1:42m	3.43s
Table	15MB	18.96s	1:39m	5.21s
Join	18MB	17.01s	1:28m	5.14s
SimpleNet	11MB	11.70s	1:57m	1.82s

Table 1: Compiler and Simulation Costs

The four numbers, in order, the RDLC and Overlog compiler memory usage and time, minus the 30MB and the time it takes to load all of RDLC and the JVM. At 150K lines of code and 10MB even for a simple Overlog design, the compiler is clearly overly large. See section 9.3 for more information.

The next two metrics are related to the performance of hardware simulations using the industry standard ModelSim SE 6.1e. The load time is measured from simulator invocation to the completion of Verilog compilation, and is a reasonable metric for the code complexity. The simulation time is the amount of real time taken to simulate 100us circuit operation, more than enough time for all of the programs to do useful work. The clock period of the simulation is 1ns, to ease the math, despite the fact that this is unrealistic. Most of the load time is inherent in the simulator we used, especially since it uses a networked licensing scheme. Load times therefore are a relative measure of complexity.

All compilations and simulations in this table were run of an unloaded P4 3GHz with 1GB of RAM.

Aside from the memory hogging inherent in RDLC, most of which is due to inefficiencies in the RDLC2 core, and it's java implementation, these numbers are reasonably promising. We revisit the simulation time in section 9.3 however.

7.2 Micro-benchmark Performance

Shown in table 2 are the results of the Xilinx FPGA Place and Route tools for each test program. These results were produced with Xilinx ISE 8.1 on an AMD Opteron 146 with 3 GB of memory. For these we used the included XST synthesis tool, rather than the more powerful Synplify Pro because of issues with IP licensing for Xilinx specific cores.

Test	#LUT	#FF	Clock	Time
Facts	468	450	175 MHz	1 m 35 s
Simple	1155	867	172 MHz	2 m 1 s
MultiStream	2260	1546	173 MHz	3 m 47 s
Table	2303	1655	173 MHz	3 m 25 s
Join	2606	1837	177 MHz	3 m 50 s
SimpleNet	980	806	176 MHz	1 m 55 s

Table 2: Hardware Statistics

The LUT and Flip-Flop counts are presented for two reasons, first, they impose a hard limit on the number of nodes which can be implemented in an FPGA. In addition they affect the PAR tool run time and clock frequency the synthesized circuits can run at.

Given that the largest tests implemented roughly a single Overlog rule, a Virtex2Pro 70 could hold roughly a single Chord node. However, we believe that these hardware costs could be significantly reduced with better compiler optimizations.

The most impressive numbers in table 2 are undoubtedly the clock frequencies. A 100MHz design on a Virtex2Pro is fairly standard, and not too difficult, but normally anything over this must be hand optimized for performance. From the fact that RDL designs with small units generally balance LUT and Flip-Flop usage, it is clear that these designs are highly pipelined, with increases their operation frequency into a range unheard of for automatically generated designs.

As impressive as the raw numbers in table 2 is the simple fact that P2 takes ~100ms to respond to a query, whereas our hardware implementation will typically respond in maybe 100 clock cycles, which at 100MHz results in a 1us turn around on input queries, a very impressive result for a such a high level input language as Overlog. Of course the price for this performance includes the cost of owning and FPGA board, which many researchers do not.

However the biggest price, is undoubtedly the cost of recompiling a system through the FPGA synthesis

tools. While the runtimes for these projects are reasonable, many larger BEE2 applications have been known to take in excess of 12 hours, implying that progress in this area will be required to make hardware Overlog implementations as flexible as their software counterparts.

8. Conclusion

By and large, this project marks a considerable success and the confluence of several research projects. By themselves Overlog, RDL and the BEE2 are all interesting, but combined they promise to open up both new research avenues and new application areas. Despite the successful execution of this project, there exists a significant amount of work to be done, as outlined in section 9. In the remainder of this section we detail the lessons from our implementation efforts, and discuss their impact.

Many of the implementation decisions in this paper relate to the running time and space of our design. When switching from software to hardware, O(1) becomes O(n) because operations reflect their per-bit cost in the absence of a fixed bit width CPU. This caused us consternation during the design process, as we tried to optimize all of our operations, with little regard to relative run time of hardware and software. What's efficient in hardware is not the same as what is efficient in software and a project, like this, which spans the two can be tricky to design. In the end we took the view that a functional result was the primary goal and speed could wait. We feel justified in this, as it was often unclear what was feasible and what was not, and with 150 thousand lines of code in the various compilers, it has been a significant effort to get to this point. Furthermore the hardware implementation by virtue of its specialization is faster than most software could ever hope to be, even without heavy optimization.

In the end the biggest drawback of the current implementation of the Overlog compiler and language relates to its inability to handle system level I/O. While our test platform provides a clean link to software which can inject and read back tuples, the process of making this connection to an I/O block needs to be automated in the compiler. We believe this will prove to be one of the main requirements for useful systems written in Overlog, just as the ability to generate non-channel connections was key to making RDL a useful language for this project.

The best news at the conclusion of this project is the relative ease with which it was completed. Normally any hardware design this large might take a man-year or more. However we implemented it in 3 months, including 2 and ½ man-months of coding for RDLC2, and the FLEET [18, 19] compiler, an extension to RDLC2 similar to the Overlog compiler, which provided good debugging tests for RDLC2. The Overlog compiler and elements themselves took about 3 weeks and we were able to implement the complete system in about 6 man-weeks.

Furthermore, those Verilog modules which are generated by Java plugins for RDLC are very powerful, allowing the Tuple Field Processor and Reorder units described in section 5 to be built from scratch in about 2 days total, with another half day of debugging. Considering that these units amount to a small data cache and processor, along with an assembler and processor builder, this is an almost unheard of time frame.

We believe the modularity of RDL, combined with the high-level semantics of Overlog contributed significantly to the ease of development. For example, the testing was quite laborious until we implemented the Base Facts unit to supply raw tuples specified in an Overlog program to a P2 system at startup. What's more it took only a few hours to add all of the language and compiler support for this feature.

Overall the success of these compiler tools in assisting their own development suggests that they are most definitely useful, and we look forward to building real applications with them. This success has also been a large part of our interest in applying these tools to architecture debugging for the RAMP project.

9. Future Work

A significant fraction of the time to complete this project was simply getting to the point where sufficiently complicated RDL could be compiled to hardware. In the end, the quality of implementation of the Overlog compiler has been sacrificed somewhat for speed of development: there's a definite lack of flexibility in the RDLC2 framework and supported Overlog constructs.

For example a better framework for the planner would easily allow us to implement Chord, as outlined below, as well as providing compile time optimizations like constant propagation and expression simplification.

9.1 Chord in Hardware

In the relatively short term we hope to be able to boot a chord ring in hardware. By adding type declarations, and some minor lexical and syntactic changes, we were easily able to compile the Chord specification used for P2 testing, but it cannot be transformed into RDL with the current version of RDLC2 and the Overlog planner.

The first big problem here is the use of null or out-of-band values. We did not implement support for these well enough, and changing this would have required rewriting large portions of Java that generates Verilog. Furthermore compiling Chord also exposed a latent bug in our handling of nested table scans for rules with multiple materialized table inputs and again the fix for this would have involved painful surgery on Verilog. We will discuss these problems in section 9.3.

9.2 Java Implementation

Early in this project, while we were using RDLC1, which has Java output support, we actually did a fair amount of work on a java implementation to match the hardware. Because RDLC is designed generate code for software and hardware hosts equally easily, this would open up interesting possibilities for research. Firstly, this would provide option to split the implementation between hardware and software. Second because an RDL software system is essentially a highly parallel program with a user level scheduler, this would promote a whole range of systems research from schedulers to protection and IPC.

The main different would be a lack of TFP in software, as having access to the RDLC java generation back end would allowed us to hard code the tuple operations with the need for a TFP or PEL transform.

9.3 RDLC3

Many of the problems we encountered were related to shortcomings of the RDL Compiler and the compiler framework. While the second generation code base greatly increased our capabilities the actual compiler code itself is still rather messy, making changes to plugin compilers, like Overlog, difficult.

Our biggest problem was the immaturity of the library of hardware units upon which we could draw. Tools like the Xilinx Core Generator already exist in this area but are vendor specific, do not fit the RDL model and are tend to be both buggy and closed source. One of the features on the short list for RDLC3, and key to our implementation of Chord is better integration of generated units into the compiler either in the form of macro-replacement, language fragments [20] or at least a better Verilog output back end.

Expanding the unit library to include a DRAM controllers, as opposed to just the SRAM generators which we built, would allow us to build a paging mechanism to store of large relations in the DDR2 DRAM on the BEE2. DRAM controllers as mentioned in section 2.4, remain one of the most time consuming pieces of hardware and yet they will be required to produce large scale Overlog systems which go beyond simple overlay networks.

9.4 Debugging Tools & Features

From the 2-3min turn around time on debugging even our micro benchmarks, it became clear that we need better performance out of the simulation environment for those situations where an FPGA is a poor test platform.

In addition, we faced a significant number of crash bugs in ModelSim during the course of RDLC2 development. Some were alleviated by an upgrade, but some have been documented by the authors for up to two years now without a forthcoming fix.

In addition to finding a better simulation environment, we believe the by developing on the ideas in [16] and section 3.2 we could more easily debug Overlog, the compiler itself and general RDL designs. As part of RDLC3 we plan to integrate the Overlog compiler with the forthcoming RDL debugging framework to support debugging not just of Overlog designs, but of all RDL designs.

We believe these enhancements, experience and maturity in our tools with lead to their use in real systems in short order, as they provide much needed functionality.

10. References

1. Loo, B.T., et al., *Implementing Declarative Overlays*. 2005: UC Berkeley. p. 1-16.
2. Gibeling, G., A. Schultz, and K. Asanovic, *RAMP Architecture & Description Language*. 2005, UC Berkeley.
3. Kohler, E., et al., *The Click modular router.* ACM Transactions on Computer Systems, 2000. **18**(3): p. 263-97.
4. Kohler, E., R. Morris, and C. Benjie. *Programming language optimizations for modular router configurations*. in *Tenth International Conference on Architectural Support for Programming Languages and Operating Systems. San Jose, CA*. 2002.
5. Loo, B.T., J.M. Hellerstein, and I. Stoica. *Customizable Routing with Declarative Queries*. in *Third Workshop on Hot Topics in Networks (HotNets-III)*. 2004.
6. Loo, B.T., et al., *Declarative routing: extensible routing with declarative queries.* SIGCOMM Comput. Commun. Rev., 2005. **35**(4): p. 289-300.
7. Szekely, B. and E. Torres, *A Paxon Evaluation of P2*. 2005.
8. Stoica, I., et al. *Chord: a scalable peer-to-peer lookup service for Internet applications*. in *ACMSIGCOMM 2001 Conference. Applications, Technologies, Architectures, and Protocols for Computer Communications. San Diego, CA*. 2001.
9. Culler, D.E. and Arvind. *Resource requirements of dataflow programs*. in *Honolulu, HI*. 1988.
10. Chang, C., J. Wawrzynek, and R.W. Brodersen, *BEE2: a high-end reconfigurable computing system.* IEEE Design & Test of Computers, 2005. **22**(2): p. 114-25.
11. Wawrzynek, J., et al., *RAMP Research Accelerator for Multiple Processors*. 2005.
12. Huebsch, R., et al., *Querying the Internet with PIER*. 2003. p. 1-12.
13. Rodriguez, A., et al. *MACEDON: methodology for automatically creating, evaluating, and designing overlay networks*. in *First Symposium on Networked Systems Design and Implementation (NSDI '04). San Francisco, CA*. 2004.
14. Patterson, D., *Research Accelerator for Multiprocessing*. 2006.
15. Droz, P.-Y., *Physical Design and Implementation of BEE2: A High End Reconfigurable Computer*, in *EECS*. 2005, UC Berkeley: Berkeley, CA.
16. Singh, A., et al. *Distributed Monitoring and Forensics in Overlay Networks*. in *Conference of the European Professional Society for Systems*. 2006. Leuven, Belgium.
17. Manish, V., N. Vachharajani, and D.I. August. *The Liberty structural specification language: a high-level modeling language for component reuse*. in *2004 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'04). Washington, DC*. 2004.
18. Sutherland, I., *FLEET – A One-Instruction Computer*. 2005. p. 1-12.
19. Coates, W.S., et al. *FLEETzero: an asynchronous switching experiment*. in *Proceedings Seventh International Symposium on Asynchronous Circuits and Systems. ASYNC 2001. Salt Lake City, UT*. 2001.
20. Adams, S.R., *Modular Grammars for Programming Language Prototyping*. 1991, University of Southampton.