The Reconfigurable Streaming Vector Processor (RSVP™)

Silviu Ciricescu, Ray Essick, Brian Lucas, Phil May,
Kent Moat, Jim Norris, Michael Schuette, and Ali Saidi†
Motorola Labs, Motorola, Schaumburg, IL
The Mitre Corporation, Bedford, MA†
phil.may@motorola.com

Abstract

The need to process multimedia data places large computational demands on portable/embedded devices. These multimedia functions share common characteristics: they are computationally intensive and data-streaming, performing the same operation(s) on many data elements. The Reconfigurable Streaming Vector Processor (RSVP™) is a vector coprocessor architecture that accelerates streaming data operations. Programming the RSVP architecture involves describing the shape and location of vector streams in memory and describing computations as data-flow graphs. These descriptions are intuitive and independent of each other, making the RSVP architecture easy to program. They are also machine independent, allowing binary-compatible implementations with varying cost-performance tradeoffs.

This paper presents the RSVP architecture and programming model, a programming case study, and our first implementation. Our results show significant speedups on streaming data functions. Speedups for kernels and applications range from 2 to over 20 times that of an ARM9 host processor alone.

1. Introduction

Portable/embedded devices will find their way into many varied products in the future. These products will include image/video capture devices (image finishing), and portable computation/communication devices (handwriting recognition, voice recognition and synthesis, and graphics). The underlying algorithms for these tasks share common characteristics. These characteristics arise because the data that is being processed is streaming in nature.

What we mean by streaming is that the data is produced/acquired as a stream of elements, each of which is relevant for a short period of time, and each of which undergoes the same computation or set of computations. When these computations are complete, the result is stored/displayed, and the data elements that fed this calculation are not used again. The characteristics of the data stream are that elements have a high degree of spatial locality, but relatively poor temporal locality.

In addition to having basic spatial locality, the data access patterns for these applications is such that entire input and output sets can be completely described prior to the calculation. These characteristics allow prefetching of data ahead of the computation, thus hiding memory latency. What's more, these access patterns can be described by a small set of characteristics. These simple, intuitive descriptions define arrays of data elements or vectors.

The problem that we solve with the Reconfigurable Streaming Vector Processor (RSVP) is: how to efficiently process streaming vector data while at the same time presenting a simple, intuitive programming model. The solution we arrived at was to design:

- A coprocessor to operate synchronously with an existing host CPU
- A programming model that separates the description of data from computation
    - Data described by location and shape in memory
    - Computation described by data-flow graph [1]

The location of the data vectors is a pointer (address), and vector shape is a simple rule for calculating the next address given the current element address. The data-flow graph describes a set of ordered, dependent computations applied to each element in the vector. These straightforward descriptions provide the ease of programming desired, while at the same time enabling a wide range of implementations having different cost/performance tradeoffs.

We have implemented a complete system on chip (SoC) including an ARM9 processor and an RSVP processor. In addition, our chip contains a complete set of peripherals for multimedia system design. In support of this chip, we provide a set of APIs for programming the RSVP processor, a complete software development tool-chain, a set of libraries for commonly used

Proceedings of the 36th International Symposium on Microarchitecture (MICRO-36 2003)
0-7695-2043-X/03 $17.00 © 2003 IEEE

multimedia kernels, and a multimedia system development board. This complete SoC package is targeted at low-power portable/embedded multimedia platforms. The chip is fabricated in TSMC 0.18um CMOS technology, and will be discussed in more detail in the Implementation section of this paper.

The next section presents the motivation for the RSVP design. Following this is a description of streaming computation, the RSVP programming model, and the architecture it implies. Next, a description of the first RSVP implementation is discussed. Finally, benchmark results are presented for both kernel-level code and complete applications showing significant speedups.

2. Motivation

Vector machines have played an important role in the supercomputer arena for over 25 years [2]. Recently a resurgence of interest in a type of vector processing architecture has occurred, but targeting multi-media rather than the traditional problem domain of the supercomputer. These architectures are exemplified by the MMX extensions to the Intel IA32 architecture [3,4], and the AltiVec extensions to the PowerPC architecture [5,7]. In the design of the RSVP architecture, we've taken an approach that differs from this current trend because it results in an architecture that is more efficient, is capable of a wider range of compatible implementations, and is easier to program.

Most of these recent vector architectures define a RISC-like, load-store programming model with wide, fixed-sized vector registers. These "wide-word SIMD" (WW-SIMD) machines offer a set of instructions that perform operations on all the "vector" elements in a register independently and in parallel. The level of abstraction of these architectures is low, and to achieve maximum performance with the WW-SIMD approach is a difficult programming problem. The burden is on the programmer to tune the code and data so that it matches the size, alignment, and memory characteristics of each implementation of the architecture. Another consequence of the low architectural abstraction level is the difficulty in scaling the WW-SIMD architecture to produce implementations at different price/performance points. Additionally, WW-SIMD machines produce speedups on benchmark code that are less than their wide maximum issue-width suggests [6].

The increasing performance gap between memory and processing is something else designers must contend with. The earliest memory-to-memory vector architectures [8], allowed designers to hide the mismatch between memory latency and computation delay prevalent in the technology of the day (magnetic cores). The application of hierarchical memory systems caused later vector machine designs to follow the RISC register-to-register trend [9,10]. Currently, however, the gap between memory latency and processing has been increasing due to fast on-chip ALUs and slow (relatively speaking) off-chip memory [11]. Memory designers have done a fantastic job of increasing the bandwidth of modern memory systems (SDRAM, DDR, RDRAM), but the latency of memory is constrained by natural laws. For this reason, much recent research has gone into the area of streaming architectures [12,13,14,15,16].

3. A Streaming Computation Model

The RSVP architecture utilizes a stream-oriented approach to vector processing, which is described in this section. Our architecture decouples and overlaps data access and data processing and eliminates the need for programmers to explicitly schedule memory accesses. We define several independent load/store units, which take advantage of the nature of multimedia data. They prefetch vector data from long-latency, wide memory and turn it into narrow, high-speed streams of vector elements, which communicate with the processing units via interlocked FIFO queues. A consequence is that vector alignment and size, and memory access scheduling (issues in WW-SIMD machines) become irrelevant to the programmer.

A streaming architecture offers three opportunities for improving performance by increasing parallel processing:

- Decoupled operand fetch
- Deep pipelining (function unit chaining)
- SIMD processing

These opportunities can be applied independently and concurrently.

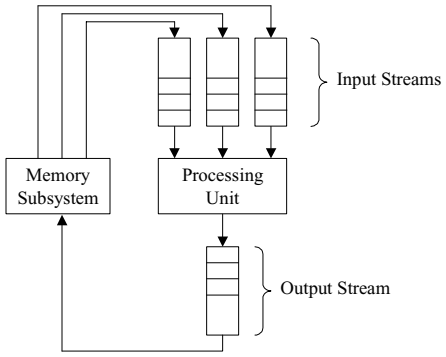


Figure 1, Decoupled operand prefetch allows data to be fetched ahead of the computation.

The decoupling of operand fetches separates the data access and data processing into independent units that operate asynchronously. The operand access units, called vector stream units (VSUs), and the processing unit, communicate via interlocked FIFO queues (see Figure 1).

0-7695-2043-X/03 $17.00 © 2003 IEEE

This allows the stream units to take advantage of available memory bandwidth to prefetch the data and have it ready for processing before it is needed. This form of dynamic scheduling is more effective and simpler than static scheduling performed by a programmer or compiler.

AltiVec has stream units, which implement a decoupled operand prefetch to cache. However, the operands must still be moved from cache to a register by explicitly programmed load instructions, and prefetching doesn't guarantee the data will be in the cache when fetched [17].

Deep pipelining is possible because instructions in the vector loop are treated as a data-flow graph of interconnected stream and function units. In a fully pipelined implementation, each instruction is mapped to a function unit, and each operand is mapped to a communication resource, with one or more results produced every clock cycle (after a pipeline fill interval). Our RSVP pipeline implementation splits the Processing Unit block in Figure 1 into an N-stage pipeline by chaining multiple function units together. This allows RSVP implementations with higher clock frequencies and increased resource utilization.

In addition, many vector operations work on elements of the datastream independently, providing additional opportunities for parallelism in the form of SIMD processing. Placing multiple taps on each of the stream units allows vector elements to be processed in SIMD fashion. This is similar to WW-SIMD architectures with one important difference. In WW-SIMD machines, the amount of SIMD parallelism is determined ahead of time by the width of the programmer-visible SIMD registers. In the RSVP architecture the speedup is limited only by resource limitations, algorithm characteristics, and dataset (vector) size. RSVP programs are specified in a way that allows scheduling tools to exploit both pipeline and SIMD parallelism concurrently.

Additionally, since the code for a streaming architecture is independent of any fixed hardware data size, the same program can execute on a wide range of implementations, providing the potential for binary compatibility and ease of software reuse.

Finally, the RSVP architecture is an enhancement to an embedded general-purpose processor (i.e., the host). It is a "co-processor" that operates synchronously with the host processor instruction flow, presenting a familiar single-core programming model. It augments the data processing instructions of that processor, but does not participate in the control flow. Therefore an RSVP processor depends on the host processor for conditional branches, subroutine calls, etc. The RSVP architecture does provide a mux-like select operation for simple predicated execution of dataflow graphs. Most microprocessors can be used as a host to an RSVP processor.

4. Architecture and Programming Model

In creating the RSVP programming model, we chose a model for which it was easy to express parallelism at the level of "C" operators. The natural choice was Synchronous Data-flow (SDF) graphs [1] with DMA-like sources and sinks. Programming the RSVP architecture consists of describing the input and output vectors and scalar values for a particular computation, and describing the computation itself as a data-flow graph.

4.1. Describing Data

RSVP data is described as scalar values, accumulator initialization values, and vectors. Figure 2 shows the programmer-visible data description registers in the RSVP architecture. These registers are accessed via host coprocessor instructions mapped to the RSVP processor.

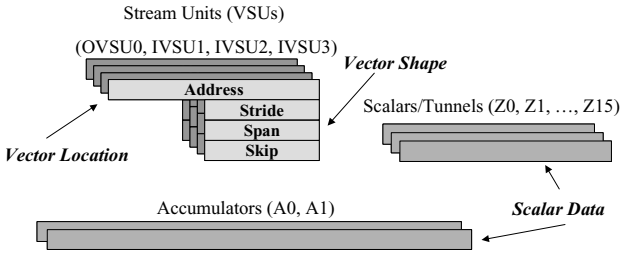


Figure 2, Describing RSVP data involves setting accumulator and scalar values, and describing the location and shape of vector data in memory.

Vectors to be processed reside in memory, and are handled by vector stream units (VSUs) in the RSVP architecture. Their description consists of a pointer to the first element in each vector and a description of the vector shape. The shape of the vector consists of three scalar values: *stride*, *span*, and *skip*. *Stride* describes the spacing between each fetched/stored element (inclusive of the element). *Span* describes how many elements to fetch/store at stride spacing before applying the second-level skip offset. *Stride* and *skip* may be positive or negative, but *span* is always positive.

This set of parameters allows the programmer to: describe a two-dimensional sub-array within a larger two-dimensional array, uniformly sub-sample an array, and create a circular (modulo) access to memory using a negative skip value. Some examples of this are shown in Figure 3.

Scalar data for the RSVP architecture is defined to be any data that is loop invariant, and is described by its value. Two 64-bit accumulators and 16 32-bit scalar/tunnel registers make up the programmer-visible

scalar state in the RSVP architecture (tunnels will be described in more detail in the following section). Scalar values may also be specified as immediate values in an RSVP program.

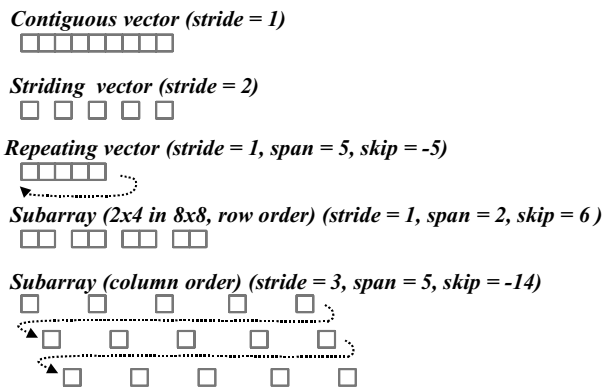


Figure 3, The *skip*, *span*, and *stride* shape descriptors allow a range of vector shapes to be specified.

The above structures are accessible via a C API that we have implemented for an RSVP processor plus an ARM9 host processor. This API hides the host processor-to-RSVP communications details from the programmer, and is implemented as inline assembly functions that execute load/store or coprocessor instructions depending upon the registers being accessed. This reduces the host setup overhead while still providing a comprehensive interface. Examples of this API are shown below.

- VSU setup examples
  - specify address of an input vector of byte elements
    ```
    void _vibyte( int vsu_num, void *addr);
    ```
  - specify output vector shape
    ```
    void _voshape( int vsu_num, short stride, unsigned short span, short skip);
    ```
- Accumulator and constant interface examples
  - get/set a scalar/tunnel value
    ```
    void _vsetz( int scalar_num, long val);
    long _vgetz( int scalar_num);
    ```
  - clear the contents of an accumulator
    ```
    void _vclra( int acc_num);
    ```
  - get an accumulator value shifted and saturated to a word
    ```
    long _vgetaw( int acc_num, int shift_amt);
    ```

4.2. Describing Computation Using Dataflow Graphs

RSVP programs use a "Data-flow Graph" (DFG) language to express vector operations in a machine-independent manner. In this DFG language, all dependencies are explicitly stated to facilitate parallel execution. Each node in the DFG is denoted by a descriptor, which specifies:

- Input operands. The input operands are specified as relative references to previous nodes rather than named registers. This feature helps eliminate the unnecessary contention for named registers as well as the overhead associated with register re-naming.
- The operation to be performed by the node.
- The minimum precision of its output value. This can be derived from the precision of the input operands and from the operation performed by the node. However, implementations are allowed to use more precision if that is easier.
- The signedness of the node.

In addition to data dependencies, the data-flow graph may also express limited iteration-to-iteration dependencies. The results that are passed between iterations are indicated by node descriptors that access the accumulators or a small set of named FIFOs called "tunnels". Tunnel nodes save the result of an operation in the current iteration while providing the result produced in the previous iteration (i.e., the source and the sink of the data-flow is the same). This greater overlapping of multiple iterations since the data from one iteration can be efficiently passed to the next iteration. In the case of loop unrolling, tunnels between unrolled iterations become dataflow graph arcs, with tunneling occurring only between the larger unrolled iterations. The same is not true for DFG nodes accessing the accumulators, because the source and the sink accumulator nodes are separate and are located at different points in the DFG limiting the degree of iteration overlap.

Order dependencies are the last type of dependencies that might be present in a data-flow graph. This type of dependency is present when multiple node descriptors refer to the same VSU (i.e., when multiple elements of a vector are processed in one iteration, there is an implied order between successive queue access operations). The sequential execution of the nodes that form the DFG provides a reference result, which must be matched by any parallel execution of the DFG nodes on any implementation. This sequential list of nodes is known as the linear form or linear DFG. Simply stated, the linear form of the DFG is an ordered list of node operators, the sequential execution of which defines the DFG's behavior.

The DFG is mapped onto our current hardware implementation by a micro-architecture aware scheduler, which is part of a larger set of tools that form our DFG compiler. An example of a linear DFG appears in a following section (Quant Programming Example).

The API for the RSVP processor supports execution of data-flow graphs. The programmer must specify the

location of the DFG in memory and the number of iterations that the DFG should execute (after having setup the data descriptions). The RSVP processor responds by taking control of execution flow and memory, retrieves the specified DFG from memory, and executes it for the specified number of iterations. When it has completed, control is returned to the host processor. Examples of this API are shown below.

- load DFG at graph_addr
  `void _vload( void *graph_addr);`
- execute previously loaded DFG for cnt iterations
  `void _vrepeat( int cnt);`
- load DFG at graph_addr and execute for cnt iterations
  `void _vloop( void * graph_addr, int cnt);`

4.3. Scheduling and Binary Compatibility

The binary form of the linear data-flow graph is the form that all implementations of the RSVP architecture must execute. For many RSVP implementations, however, there will be a binary form that is more suitable for direct execution on the hardware, and that form may not be the linear DFG. For these cases the RSVP architecture specifies "universal fat binaries" (UFBs).

UFBs contain more than one binary form of a data-flow graph. UFBs may contain, in addition to the linear form of the DFG (which is mandatory), one or more optimized forms. Our DFG compiler, which has complete knowledge of the underlying hardware, creates these optimized forms. The input to this compiler is the linear DFG form. All forms contained in the UFB are structured as a linked list with the linear DFG form appearing last in the list. This list is walked by the RSVP hardware, and the first instance that it can execute is used (each list element is marked with an implementation ID). If no custom instances are found, the linear DFG form is used as a default. One way to achieve this is to employ a just-in-time compiler that allows direct execution of RSVP dataflow graphs at the system level.

4.4. Quant Programming Example

As an example of the transformation of an operation from the original C code to an RSVP program, and for an example of linear DFG programming, consider the "Quant" algorithm. Quant compresses video images through quantization. It appears in multiple standards such as: JPEG, MPEG, and H.263. It is a good candidate for execution on an RSVP processor because its input is a vector and each element can be independently processed. Figure 4 shows the C code for Quant.

As can be seen from the C code, Quant consists of a small number of calculations followed by a loop that is executed for n iterations. To implement Quant using an RSVP processor, the host executes the preliminary calculations and the RSVP processor executes the loop. Figure 5 shows the code to execute the preliminary calculations and to set up and initiate execution on the RSVP processor. Lines 1-4 are identical to the original C code. Lines 5-8 utilize the API calls described above. In this case, the API is used to initialize input VSU 1 with the address of the input vector, initialize output VSU 0 with the address of the output vector, load the scalar registers with the values of b and rq, set the iteration count, and initiate execution on the RSVP processor. The stride, span, and skip parameters of the VSU's are not set. The default values for these parameters are used, which configure the VSUs to access a one-dimensional contiguous vector.

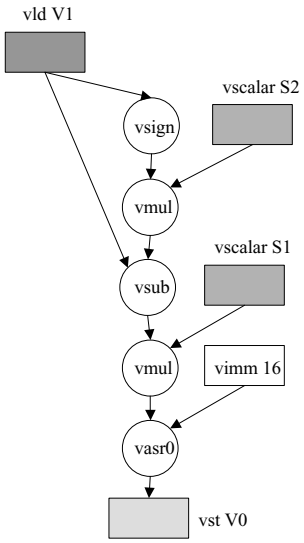
```
void quant(short *out, short *in, int n,
short qp)
{   long rq, b, c;
    rq = ((1 << 16) + qp) / (qp << 1);
    b = qp - !(qp & 1);
    while (--n >= 0)
    {   c = *in++;
        if (c < 0)          c += b;
        else if (c > 0)     c -= b;
        *out++ = (c*rq) / (1 << 16);
    }
}
```

Figure 4, Original quant routine written entirely in C.

```
void quant(short *out, short *in, int n,
short qp)
{   long rq, b, c;
    rq = ((1 << 16) + qp) / (qp << 1);
    b = qp - !(qp & 1);
    _vihalf1(in);
    _vohalf0(out);
    _vset(1, rq);
    _vset(2, b);
    _vloop(&rsvp_quant, n)
}
```

Figure 5, Rewritten routine setting up the RSVP processor to execute the inner loop using our provided API.

Figure 6 shows the equivalent data-flow graph for the operation performed within the loop and the linear form of the DFG, which is an ordered sequence of node descriptors for the DFG (the current RSVP toolset operates on this syntax). The first descriptor, Q1, makes the next element in the input VSU queue its output and, thus, available for access by the other descriptors. Q2 tests the output of Q1 and outputs 1 if it is >0, 0 if equal to 0, and -1 if it is <0. Q3 & Q4 make the scalars 1 and 2 (b and rq) their output. The next four descriptors perform the equivalent of the setting of c based on the sign of the input vector element, multiplication of c by rq, and shifting of that value. Q10 places the result of Q9 in the output queue of output VSU 0. This completes a single iteration of the loop.




```
rsvp_quant:
Q1:  vld.s16     (v1)    //c = *in++;
Q2:  vsign.s16   Q1
Q3:  vscalar     s2      //s2 is b
Q4:  vscalar     s1      //s1 is rq
Q5:  vimm        16
Q6:  vmul.s16    Q2,Q3   //if (c<0) c+=b;
Q7:  vsub.s16    Q1,Q6   //else if (c>0) c-=b;
Q8:  vmul.s32    Q7,Q4   //c *= rq;
Q9:  vasr0.s16   Q8,Q5   //*out++ = c/(1<<16);
Q10: vst.s16     Q9,(v0)
```

Figure 6, The loop code is described in an intuitive data-flow graph form (linear (text) form and graphical form shown).

The speedups afforded by the RSVP architecture are realized by scheduling the linear form of the DFG (Figure 6), optimizing the execution to the implementation. For our first RSVP implementation, we provide a DFG compiler for this task. After compilation, the DFG in Figure 6 executes as a pipeline, producing a result every clock cycle. Each of the nodes in Figure 6 executes every clock cycle, but each is processing data from a different iteration. This iteration/operation mapping is shown in Figure 7, with the stage numbering indicating in which pipeline stage the instruction has been scheduled.

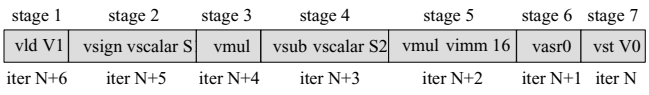


Figure 7, A DFG compiler schedules the operations as a deep pipeline of chained function units, producing one result every cycle.

4.5. Architecture

The RSVP architecture dictates a required set of behaviors for any RSVP implementation, but does not mandate a particular implementation. Software written for the RSVP architecture will be able to run on any implementation that satisfies these requirements. It allows designers the freedom to optimize for their unique goals while preserving the ability to re-use application software. The architecture that the preceding descriptions imply is diagrammed in Figure 8.

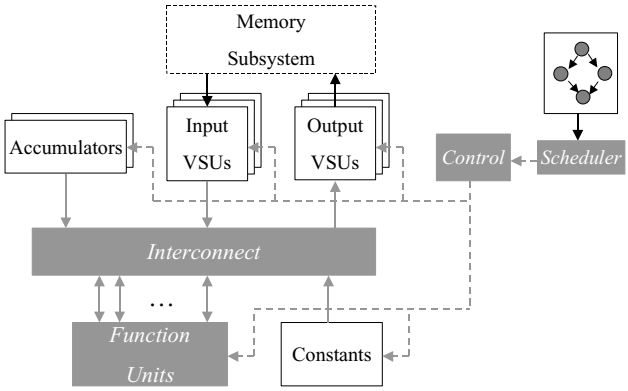


Figure 8, The RSVP architecture (white solid boxes), and supporting structures hidden from the programmer (gray boxes).

The white blocks are the programmer-visible structures described previously. The gray blocks represent the data-flow graph computation structures, which are implementation dependent and hidden from the programmer.

The components labeled Input VSUs and Output VSUs are visible in the programming model as described in the previous section. The number of output VSUs and input VSUs is defined architecturally to be a minimum of one and three respectively, and maximum of 64 each. Their responsibility is to load/store elements of the input/output vector for which they are configured. Each VSU handles all issues regarding loading/storing of the data to/from the host memory subsystem and presents the data to the datapath as elements in a queue. This means a VSU must handle issues associated with byte alignment and byte ordering. Each VSU converts memory accesses done in the width of the host's memory bus to/from a number of elements in its queue. Finally, the output VSUs must flush any data in its queue at the time of loop termination in order to ensure memory coherence.

Each input/output VSU performs all address generation involved in determining the location of the next vector element within the host memory, based on the parameters it was configured with by the host. This address generation must not depend on the loop count or on results generated by the computation, so that it can proceed independently. Upon termination of the loop, it must make available to the host, the address of the next element of the vector, analogous to the behavior of a pointer used to step through a vector in a C loop. This is to be done regardless of any prefetching of data that may

cause the VSU to fetch data beyond the last element used in the vector. In addition, each VSU must allow read/write access from the host to all of its other parameter-containing registers.

The data-flow graphs used to specify RSVP computations may contain no more than 256 nodes, and may "reach back" no more than 63 nodes in the linear (text) form of the DFG for its input operands. This puts a reasonable upper limit on the number of in-flight operands that an implementation is required to store, and the largest number of DFG instructions to be cached.

The number of accumulators and scalar/tunnel registers is defined architecturally to be a minimum of two and 16 respectively, and maximum of 64 each. Our first implementation of the RSVP architecture contains two accumulators and 16 scalar/tunnel registers. The Accumulators block supports the accumulators, and the Constants block supports scalars and tunnels (as described above in Describing Computation Using Data-flow Graphs).

The gray blocks in Figure 8 are not specified by the architecture. The Scheduler (our DFG compiler) converts the data-flow graph computation description into a machine-dependent form. The Control block implements the machine-dependent form of the data-flow graph (may be the direct linear form), orchestrating the actions of the Interconnect and Function Units. The Function Units must support all operations allowed by data-flow graph node descriptors, and the Interconnect must support operand passing (arcs in the data-flow graph) as well as storage of intermediate results. The RSVP architecture does not specify any named storage for intermediate results.

Additionally, there are system-level programming structures to support register protection in an RSVP implementation, setting up trap/interrupt event masks, context switching, just-in-time graph compilation, and other behavior. A debug interface also exists allowing programmers to set DFG address and iteration count breakpoints as well as allowing implementation-dependent access to additional internal structures.

5. Implementation

The first implementation of the RSVP architecture is intended to be a low-cost, low-power solution for portable/embedded devices. It is integrated into an ARM946-based SoC with a complete set of peripherals for a multimedia processor (image sensor interface, LCD controller, compact flash, etc.). This chip is a testbed for the RSVP architecture and provides our customers with a development platform for testing their algorithms. The chip is fabricated in TSMC 0.18um CMOS technology and appears in packaged form in Figure 9.

In addition, we have a complete set of software development tools for this RSVP implementation. These tools consist of:

- Compiler, assembler, linker for ARM (gnu-based)
- Compiler for RSVP linear data-flow graphs
- RSVP functional simulation library
- Performance simulator (timing, profiling)
- A set of multimedia routines tuned to our RSVP implementation
- A gdb-based debugger for ARM9 + our RSVP implementation

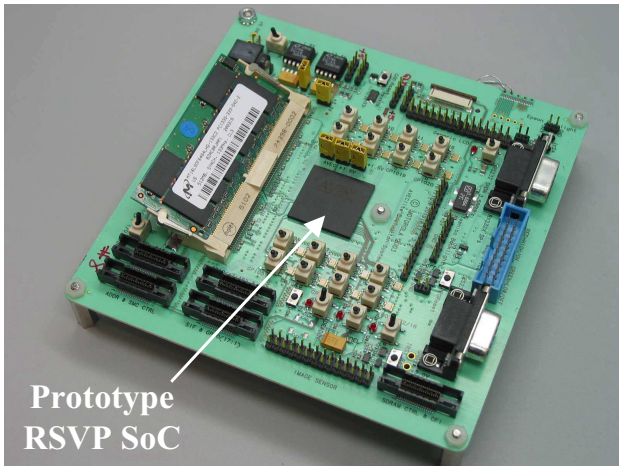


Figure 9, The first implementation of the RSVP architecture on our multimedia development board.

The structure of our implementation appears in Figure 10. Communication with the host processor is through the bus interface gasket (BIG) via either a memory-mapped AHB bus interface or the ARM coprocessor interface. The one output vector stream unit (V0) and the three input vector stream units (V1, V2, and V3) in this implementation access main memory through the BIG. A memory-mapped 8kB Tile Buffer acts as a scratch memory, and provides significant speedups to algorithms that map their data there. The Tile Buffer intercepts memory requests from the stream units if these requests address Tile Buffer memory. The Tile Buffer delivers/accepts 128-bits of data each cycle, with a latency of two cycles.

The datapath consists of a reconfigurable interconnect fabric and intermediate results store (Fabric and Queue) connecting the stream units, accumulators (Acc0 and Acc1), and constant storage (Constants & Tunnels) to a set of function units. This interconnect fabric consists of 20 links, each of which can transfer 16-bits of data from its source to its destination. These links can be reconfigured every clock cycle.

Each function unit is 64-bits wide and sliced on 16-bit boundaries. These units can operate as four 16-bit units,

one 32-bit and two 16-bit units, two 32-bit units, or one 64-bit unit. In contrast to traditional wide-word SIMD implementations, each of the slices in this RSVP implementation has its own control, so the slices constitute a MIMD unit rather than a SIMD unit (although they can be scheduled as a SIMD unit by applying the same control to all slices). All function units are fully pipelined with result latching (including the Fabric and Queue), allowing the units to be chained together to form deep reconfigurable pipelines customized to each application.

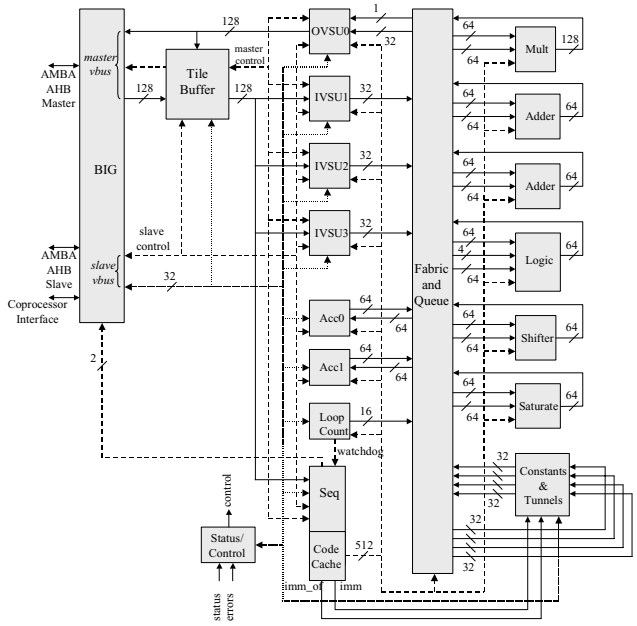


Figure 10, RSVP implementation block diagram.

For 16-bit arithmetic, this implementation provides the following resources:

- 4 multipliers
- 8 adders
- 4 logic/compare units
- 4 shift/round units
- routing and intermediate results storage
  - 20 routing links
  - 280 distributed intermediate storage locations (14/link)
- Minimum number of architecturally specified sources and sinks
  - 3 input streams
  - 1 output stream
  - 2 accumulators
  - 16 tunnel/scalar registers

We have also produced a synthesized macrocell for this RSVP implementation in TSMC 0.13um CMOS technology. The area of this core is comparable to an ARM9 core plus 16kB instruction and 16kB data caches. In effect we are doubling the area of the processor, but with a much greater than 2x increase in performance as described in the next section.

Our power simulations show that the power dissipation of our RSVP implementation is also on the same order as the ARM9 core. Since the RSVP processor operates for shorter periods of time than the ARM9 for the same task (due to speedups), the system power is reduced when the RSVP processor is employed. Since the ARM and RSVP cores dissipate the same power, and since we employ a synchronous programming model, gating the clock to one core while the other is operating provides a relative energy reduction equal to the algorithm speedup.

6. Results

We have done extensive benchmarking of our RSVP implementation as part of its design. The benchmarks were selected from application areas ranging from image processing, video, audio, and signal processing driven by customer needs. Some of the benchmarks are "kernels", small vector functions that can be completely executed within the RSVP processor. Others are "applications" for which only a portion can be executed within the RSVP processor. A description of each benchmark is given below.

Our benchmarking effort consists of running code on the ARM9 processor by itself, profiling the code, and then porting and compiling the vectorizable inner loops for the RSVP processor. The new result (cycle count) is then compared against the original ARM-only result to obtain speedups.

The following is a subset of the kernel benchmarks we have studied:

- RGB to YUV color-space conversion
- Dot product on 29 element vectors
- Finite impulse response filter
- 1024-point complex FFT
- Saxpy – add a scaled vector to another vector (400 element vectors)
- Sort – order a vector of length 1001
- Median – find median in 1001 element vector
- Discrete cosine transform (H.263)
- Quant – vector quantization (H.263)
- Dequant – vector dequantization (H.263)
- JPEG2000 wavelet transform

Figure 11 shows the speedup results for these compiled RSVP kernels. The RSVP processor versions of these kernels were run out of the tile buffer (8kB memory, 128-bits per cycle throughput) with the ARM9 processor having 4kB I and D caches (same as the SoC we fabricated). As the results show, most of these benchmarks saw a speedup of five to ten, with exceptional results for quant and dequant.

The magnitude of these speedups results from the large effective instruction issue width of our RSVP implementation. The issue width that the RSVP processor is capable of is a result of its deep pipelines of chained function units and its asynchronous load/store units. As an example, take the average issue width of the inner loop of the RGB to YUV benchmark. For ARM9, the average issue rate is 0.78 instructions per cycle, and for our RSVP implementation, the average issue rate is 9 instructions per cycle. This highlights the shortcomings of using a single-issue embedded core for multimedia processing.

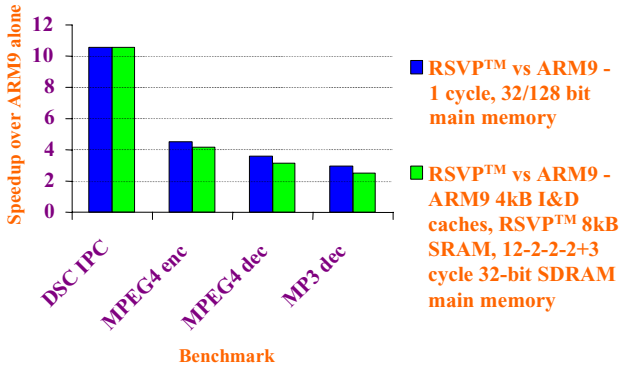


Figure 11, RSVP processor kernel speedups (operating out of tile buffer).

Even with the currently available WW-SIMD extensions for portable/embedded platforms [18,19], the processor issue width reaches a maximum of four or eight (for byte values on a few instructions). This maximum issue width, however, is not obtained for actual workloads. In fact the authors of [6] state that on average, no more than 12% of a SIMD unit's peak throughput is achieved for the Media Bench benchmark suite. This stems from the fact that in these SIMD processors, more than 75% of dynamic instructions are support/overhead related. This makes these solutions less efficient than an RSVP processor, as well as being more difficult to program.

For application benchmarking, we ran code both with SRAM memory (assuming that all data was available in single-cycle 128-bit wide memory), and with an SDRAM memory system (12-2-2-2 core-cycle burst access, with 4kB instruction and data caches for ARM9 and the 8kB tile buffer for the RSVP processor). The results highlight the ability of an RSVP processor to achieve significant speedups with realistic memory systems.

The following programs are a sampling of our application benchmarking effort:

- DSC IPC – complete digital still camera image processing chain
- MPEG4 encode and decode
- MP3 player

Figure 12 shows the speedups achieved for ARM9 + our RSVP implementation over ARM9 alone. Three of the benchmarks show speedups in the range of 2-4, with the image processing chain achieving a speedup of over 10. This speedup is due to this particular application being easy to tile, with over 90% of the load/store operations occurring in the tile buffer. The other three benchmarks use the tile buffer to a lesser extent, and additional improvements are expected from these applications with more aggressive use of tiling.

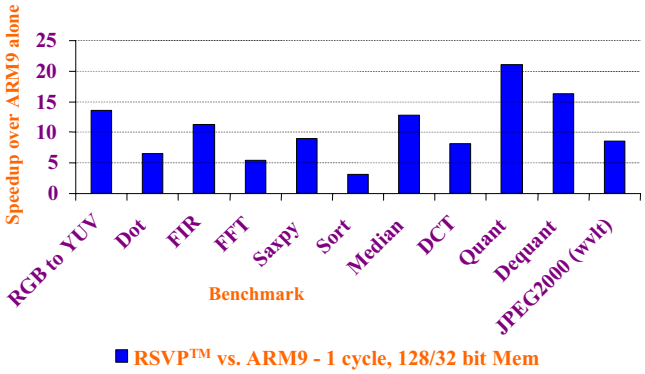


Figure 12, RSVP application speedups (ideal memory and realistic memory).

Interestingly, the performance of our RSVP implementation for these applications does not degrade appreciably when presented with an SDRAM memory system. The RSVP architecture is designed to achieve greater performance than the host processor alone with any available memory system.

7. Conclusions

The RSVP architecture is a vector coprocessor/accelerator architecture that improves the performance of general purpose CPUs on streaming data functions. It goes beyond most coprocessors/accelerators by improving time to market because of its ease of programmability, the elimination of hand-optimized assembly code, and support for software reuse through binary compatibility across multiple implementations.

The performance of an RSVP processor is much greater than that available from any embedded core alone, including those with SIMD extensions. The RSVP architecture makes efficient use of compute resources by chaining function units into deep pipelines, and the efficiency of these pipelines reduces system power.

All this is done in a way that is hidden from RSVP programmers, allowing them to focus on system functionality and algorithm development. The descriptions of computation and data are intuitive and are presented to RSVP programmers as a single-core programming model, providing a simple, familiar

paradigm that speeds software development, and reduces time to market. Support for binary compatibility enables software reuse, which supports quick time to market and third party software development. Our carefully partitioned vector coprocessor programming model and the separation of memory access from computation allows our tools to generate near optimal schedules automatically, which leads to the ability to scale up the hardware without modifying the dataflow graphs.

Finally, a complete implementation of the RSVP architecture is currently available, including ARM9-based SoC, software development tools, libraries, and a development board.

8. References

[1] E.A. Lee and D.G. Messerschmitt, "Synchronous data flow," Proceedings of the IEEE, vol. 75, no. 9, pp. 1235-1245, 1987.

[2] P.M. Johnson, "An Introduction to Vector Processing," Computer Design, pp. 89–97, Feb. 1978.

[3] L. Gwennap, "Intel's MMX speeds multimedia," Microprocessor Report, pp. 6-10, March 1996.

[4] A. Peleg, V. Weiser, "MMX Technology Extension to the Intel Architecture," IEEE Micro, pp. 42-50, Aug. 1996.

[5] L. Gwennap, "G4 is First PowerPC with AltiVec," Microprocessor Report, pp. 6-10, Nov. 1998.

[6] D. Talla, L. John, "Cost-effective Hardware Acceleration of Multimedia Applications," Proceedings of the International Conference on Computer Design, pp. 427-439, 2001.

[7] K. Diefendorff, P. K. Dubey, R. Hochsprung, H. Scales, "AltiVec Extension to PowerPC Accelerates Multimedia Processing," IEEE Micro, Vol 20, No 2, pp. 85-95, Mar/Apr 2000.

[8] R. Hintz, D. Tate, "Control Data STAR-100 processor design," In Proc. COMPCON, IEEE, 1972.

[9] R. M. Russell, "The Cray-1 Computer System," Communications of the ACM, pp. 63-72, Jan. 1978.

[10] M. C. August, G. M. Brost, C. C. Hsiung, A. J. Schiffleger, "Cray X-MP: The Birth of a Supercomputer," IEEE Computer, pp. 45-52, Jan. 1989.

[11] M. Wilkes, "The Memory Gap and the Future of High-Performance Memories," ACM SIGARCH Computer Architecture News, pp. 2-7, March 2001.

[12] C. E. Kozyrakis, S. Perissakis, D. Patterson, T. Anderson, K. Asanovic, N. Cardwell, R. Fromm, J. Golbus, B. Gribstad, K. Keeton, R. Thomas, N. Treuhaft, K. Yelick, "Scalable processors in the billion-transistor era: IRAM," IEEE Computer , pp. 75–78, Sep. 1997.

[13] U. J. Kapasi, W. J. Dally, S. Rixner, J. D. Owens, B. Khailany, "The Imagine Stream Processor," Proceedings of the International Conference on Computer Design, pp. 282-288, 2002.

[14] M. Stoodley, C. Lee, "Vector Microprocessors for Desktop Computing," Proceedings of the International Symposium on Computer Architecture, 1998.

[15] S. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, R. R. Taylor, "PipeRench: A Reconfigurable Architecture and Compiler," IEEE Computer , pp. 70–77, Apr. 2000.

[16] J. E. Smith, "The Best Way to Achieve Vector-Like Performance?," keynote presentation at International Symposium on Computer Architecture, Apr. 1994.

[17] Motorola Semiconductor, AltivecTM Technology Programming Environments Manual, Document ALTIVECPEM/D, 1998.

[18] N, C. Paver, "Intel Wireless MMX Technology," presentation at Intel Developers Forum, Sep. 2002.

[19] D. Brash, "The ARM Architecture Version 6," ARM White Paper, Jan. 2002.