Chapter 4

Merrimac: The Streaming Supercomputer

The Merrimac project aims to develop software and hardware technology to apply a stream architecture [14] to scientific computing. Our expectation, based on the first two years of the project, is that a streaming supercomputer such as Merrimac will offer more than $10 \times$ the performance per unit cost as cluster-based supercomputers while, at the same time, offering much greater capability — scalability and sustained performance. Stream processing holds the potential to improve the efficiency of scientific computing from desktop workstations to room-sized supercomputers. The design we have sketched for Merrimac can be used to build a $40K^2$ 2TFLOPS single-board workstation or a $40M$ 2PFLOPS supercomputer (composed from 1024 workstation boards).

The efficiency of Merrimac derives from its stream architecture and high-bandwidth network. A stream processor uses a deep register hierarchy to capture two types of locality, kernel locality and producer-consumer locality, that are not captured by conventional cache architectures. Capturing this locality increases the computational intensity of a program, the ratio of arithmetic operations to memory accesses. As a result, we can integrate much more arithmetic capability on each compute node for a given memory bandwidth: 128 GFLOPS per Merrimac node with 16GBytes/s (40GBytes/s peak) of memory bandwidth. Stream processors are well matched to the capabilities of modern VLSI technology where arithmetic is very inexpensive (a 1GFLOPS FPU requires 0.5mm$^2$ of chip area and 100mW of power) but off-chip bandwidth is very expensive (10GWords/sec per 150mm$^2$ chip and 1W per GWord/sec$^3$). To date converting an application to run as a stream program has involved manually restructuring the program and coding it in a stream programming language. We are optimistic, however, that this process can be automated and that vectorizable and parallelizable codes can be automatically converted to stream programs.

On codes that are inherently memory intensive, Merrimac also gains efficiency from its high memory and network bandwidth. These are features that are independent of stream processing but are not found in commercial-off-the-shelf microprocessors. Each Merrimac node includes 16 DRAM channels for a peak local memory bandwidth of 40GBytes/s (16GBytes/s on random single-word accesses). In contrast, a typical high-end microprocessor has a peak memory bandwidth of 5GBytes/s. Merrimac holds the random-access memory bandwidth flat (at 16GBytes/s) across a 16-node board, and has only an 8:1 reduction in bandwidth across a PFLOPS machine — each node can sustain 2GBytes/s of single-word random-access bandwidth to any location in a PFLOPS machine. We expect this high global bandwidth to simplify programming by reducing performance sensitivity to data placement. In particular, this will simplify load balance for irregular codes with

---

1 Merrimac is a Native American word meaning “swiftly flowing stream”.
2 These dollar figures reflect $2.5 \times$ parts cost without I/O.
3 throughout this document each word is 64 bits long
Chapter 4: Merrimac: The Streaming Supercomputer

The goals of the Merrimac project are to demonstrate the feasibility of a streaming supercomputer, to solve the key technical problems involved in building and programming such a machine, and to reduce the risk of the technology to the point where it appears attractive to industry. We do not propose to actually carry out the detailed design or construction of a streaming supercomputer. Such an effort would be of many times larger scale than our present feasibility study.

The Merrimac project has three main components: architecture, system software, and applications. The architecture effort has focused on working out the straw-man design of a streaming supercomputer node and addressing issues such as instruction-set design, register organization, cache design, and network topology. The system software effort has concentrated on the development of a stream programming system for scientific applications. This includes the development of a stream programming language, Brook, and a compiler to map Brook programs to the Merrimac architecture in an optimized manner. Our applications component has demonstrated a number of modest-sized kernels and applications on a simulated Merrimac machine. These applications serve to evaluate and provide feedback on both the programming system and the architecture.

The Merrimac program was launched in September 2001 and is now completing its second year. During the first year of the program we sketched the architecture, developed a rudimentary programming system, demonstrated a few simple applications on a simulated machine, and identified key technical issues. That effort met our goal of completing a quick feasibility test on the technology, but did not leave us with an infrastructure on which to build. During this second year of the program our efforts have focused on building a robust experimental infrastructure. We have refined the Brook language to make it both more expressive and easier to compile. We have scrapped our rudimentary compiler (based on Metacomilation) and are in the process of co-developing a new compiler, capable of deeper analyses, with Reservoir Labs. In parallel with this infrastructure building, we have addressed a number of key technical issues on both the hardware and software side, and have demonstrated more significant (3-D and irregular) applications and kernels.

While we have continued to make some progress in parallel with our infrastructure building, the compiler has become a bottleneck for both applications and architecture studies. Developing the compiler has taken longer than we expected, in part due to two false starts (using the Metacompiler, and using the open research compiler (ORC) infrastructure). As a result, many architecture studies are on hold awaiting the completion of the new compiler. Also much serious applications work cannot complete until the new compiler is stable. Overall, compiler delays have put us about six months behind our original plan. We expect a basic version of the Reservoir Brook compiler to be available soon which will remove this bottleneck.

The remainder of this chapter describes in more detail our progress over the second year of the Merrimac project.

4.1 Merrimac Architecture

The proposed Merrimac streaming supercomputer is constructed from just two custom chip types, a stream processor node, and a router. A block diagram of the proposed machine is illustrated in Figure 4.1. Each node consists of a single 128GFLOPS stream processor chip (described in more detail in Section 4.1.1) and 16 1Gbit DRAM chips (2GBytes of memory) organized as 16 separate parallel DRAM channels (40GBytes/s peak, 16GBytes/s random access memory bandwidth). This ratio of 128GFLOPS to 2GBytes of memory appears unbalanced compared to conventional machines. However, it is in fact well balanced by both cost and power. The majority of node cost and node power is consumed by the memory. Increasing the amount of memory per node would...
4.1. MERRIMAC ARCHITECTURE

decrease the performance per unit cost. The very high bandwidth to remote memory (flat on each board) reduces the impact of a small amount of memory per node. Large computations need large amounts of total memory, and high global bandwidth, not large amounts of memory per node.

Sixteen nodes (16 processor chips and 256 memory chips) are packaged together on a single board. This board also contains four high-radix router chips (not shown) that connect the processors on the board with a flat (16GBytes/s) memory bandwidth and provide a tapered bandwidth (4GBytes/s per node, 64Gbytes/s total) off the board. Each router also contains a 10Gigabit Ethernet (XAUI) channel for I/O connection, and an I/O processor. Each board has an aggregate capacity of 2TFLOPS and 32GBytes. We see this single-board unit as an attractive desk-side workstation as well as a building block for larger systems.

Up to 32 boards can be combined in a cabinet, along with 8 router boards that provide interconnection. Each cabinet (standard 19-inch rack) holds 512 nodes and has an aggregate capacity of 64TFLOPS and 1TByte of memory. Finally, up to 32 cabinets can be connected via optical interconnect to a central switch cabinet. The central switch is composed of the same routers used on each 16-node board, and for intra-cabinet switching. Such a 32-cabinet system contains 16K processors and has an aggregate capacity of 2PFLOPS and 32TBytes of memory. One attractive property of stream processors is that they can reach PFLOPS performance levels with a reasonable number of processors (8K) simplifying issues of reliability and load balance. We also expect them to sustain a larger fraction of this PFLOPS on real applications than machines based on conventional microprocessors.

It is instructive in understanding the efficiency advantage of a stream processor to compare the proposed design of Merrimac to a conventional cluster-based machine. At the node level, Merrimac offers more than an order of magnitude more peak performance 128GFLOPS vs 4GFLOPS and significantly more peak memory bandwidth 40GBytes/s vs 5GBytes/s. More importantly, on applications with producer-consumer locality, stream processing enables Merrimac to sustain a much higher fraction of peak performance by reducing the demand on memory bandwidth (effectively multiplying the already high memory bandwidth by a significant constant).

At the system level, the high radix network offers flat memory bandwidth of 16GBytes/s across a board (16 nodes) and nearly flat (8:1 local/global ratio — 2GBytes/s) memory bandwidth across the entire machine. In contrast, a typical cluster offers a global memory bandwidth of less than 100MBytes/s per node and often hides this bandwidth under significant MPI software overhead. The high global bandwidth of Merrimac, even on single word random references, enables it to sustain a high fraction of peak performance on global applications, and simplifies programming by reducing the importance of data placement.

The remainder of this section describes our work on the Merrimac architecture over the past year. Starting in Section 4.1.1 we describe the design of the Merrimac node in more detail and discuss the results of a study to establish the feasibility of constructing the node by accurately estimating its area and power. The results of a study on register organization are described in Section 4.1.2. This study shows that by providing the ability to index into the SRF, the advantages of stream locality can be realized by a broader class of applications. We have found that this capability is particularly useful with numerical codes that access data with a regular stencil pattern. Section 4.1.3 studies the use of fused-multiply add instructions in the FPUs of Merrimac and shows that with an appropriate compilation strategy they can improve performance significantly. We look at using these multiply-add units to perform operations like divide and square-root in Section 4.1.4. The architecture of the stream cache and the use of remote memory operations is investigated in Section 4.1.5.
4.1.1 Floorplan, Area, and Power Estimates for Merrimac Node

In order to assess the cost and feasibility of the proposed Merrimac node, we have performed an analysis to estimate the die area and cost of the Merrimac stream processor chip. This chip contains a pair\(^4\) of 64-bit scalar processors, a stream co-processor, a cache subsystem, a memory controller and a network interface.

Our estimates are based on an ASIC standard cell design methodology and a 90nm \([23]\) (drawn gate length) CMOS technology. The 90nm CMOS technology will be in production in 2004-5 and is a conservative technology for Merrimac which is targeted for 2006. In a typical 90nm CMOS process, the delay of an inverter with a fan out of 4 (driving 4 identical inverters) is 27ps. We refer to this unit of delay as a FO4 and express other delays in units of FO4s. The target operating frequency for Merrimac is 1GHz, or 37 FO4s. This is a very conservative number. Cycle times of 20FO4s are quite common, and very aggressive designs approach 10FO4s.

Our area and power estimates are based primarily on the area and power of corresponding units in the Imagine stream processor \([16]\), a stream processor designed at Stanford for signal and image processing. Imagine was designed in a 180nm\(^5\) standard cell process and shares many common structures with Merrimac. The area and power numbers for Imagine structures were scaled to

\(^4\)A pair of processors is used for fault detection.

\(^5\)The Imagine process has 180nm metal rules and 150nm drawn gate length. The metal rules largely determine the area and power of the processor.
account for process scaling from 180nm to 90nm and for an increase in word width from 32-bits (Imagine) to 64-bits (Merrimac). The area and power of memory arrays and register files are estimated from published data on 90nm memory arrays.

The only function-unit in the stream co-processor is a multiply-add unit that operates on 64-bit floating point and integer numbers. Reciprocal, square-root, and other related operations are performed as described in Section 4.1.4 by using a lookup table for an initial estimate and then a succession of multiplies and adds.

Floating point numbers are represented using the IEEE double-precision format (11 bits exponent, 53 bits mantissa including leading one). To deal with unnormalized numbers, inside the clusters, the exponent is extended by one bit to 12 bits and the numbers are converted between IEEE representation and extended exponent representation at the interface between the cluster and the stream register file.

The micro-architecture of the multiply-add unit is shown in Figure 4.2. In the first stage the multiplication is carried out on 64 bit mantissa although only 53 bits are used for floating point numbers. 32 partial products are produced by 2 bits Booth encoders, then those 32 partial products are reduced down to 2 through a carry save adder tree. This carry save adder tree also contains XORs necessary for the inversion of the A*B product in the case of an C-AB operation.

In the case of a floating point addition, the difference of the exponents will determine which
### Table 4.1: Multiply-Add unit timing estimates

<table>
<thead>
<tr>
<th>Unit</th>
<th>Exponent</th>
<th>Unit</th>
<th>Mantissa</th>
<th>Pipeline Stage</th>
</tr>
</thead>
<tbody>
<tr>
<td>12 bit adder</td>
<td>10</td>
<td>Buffer A to all of B</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>2 bits Booth encoding</td>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>12 bit subtract</td>
<td>12</td>
<td>Carry Save Adder tree</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>MUX</td>
<td></td>
<td></td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Shifter 0-53</td>
<td></td>
<td></td>
<td>24</td>
<td>24</td>
</tr>
<tr>
<td>MUX</td>
<td></td>
<td></td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Carry Save Adder</td>
<td></td>
<td></td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Full Adder 64 bits</td>
<td></td>
<td></td>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>Leading one detector</td>
<td></td>
<td></td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>Exponent Adjust</td>
<td>12</td>
<td>Shifter 0-53</td>
<td>24</td>
<td>36</td>
</tr>
</tbody>
</table>

Operand needs to be shift aligned and the shift amount. Both partial products need to be shifted in case that the A*B product needs to be shifted. This second shifter structure is necessary because we don’t want to do the full carry propagating addition until the end.

For an integer operation, the addition is performed on the lower 64 bits of the 128 bit result (AB0 low and AB1 low) of the multiplication as opposed to the upper 64 bits for a floating point multiplication followed by an addition.

Next the 2 partial products and C are added by a carry save adder followed by a full adder. At this point the result of a floating point multiplication and all integer operations will be available. Finally for a floating point addition of multiply-addition, the mantissa needs to be normalized and the exponent adjusted.

Table 4.1 estimates the time taken by each unit and the pipeline stages assuming a 1 GHz clock at 37 FO4 per clock cycle in 90 nm technology. Floating point multiply, and all integer operations take 3 clock cycles while floating point addition and multiply-add take 4 cycles. The compiler that schedules instructions on the clusters, adapted from the Imagine kernel scheduler, takes advantage of operations with different latencies to optimally schedule kernels on the stream co-processor.

Figure 4.3 shows the Merrimac floorplan and the details of the clusters. Each cluster contains four multiply-add units (FPUs), their associated local register files (LRFs), and one bank of the stream register file (SRF). Scaling from the Imagine FPUs, each Merrimac FPU is expected to be 0.9mm × 0.6mm (about 0.5mm²). Each bank of the SRF is integrated into its associated cluster to reduce latency and power dissipation. The SRF bank attached to each cluster has a capacity of 8k words (a total of 128K words across 16 clusters), and includes stream buffers to interface the SRF with the register file and the memory controllers. The intra-cluster switch that connects the FPUs, LRFs, and SRF within a cluster is routed on upper metal layers over the top of the FPUs and LRFs. Adding the four FPUs, the SRF, the register files, and overhead, each cluster is expected to measure 1.6mm × 2.3mm (3.7mm²).

The LRFs are currently organized as three two-port (one read, one write) 64-word register files per FPU — one LRF for each of the three inputs of the FPU. Each register file has a read port connected to an FPU input and a write port that accepts values from the intra-cluster switch. These values may come from any ALU output, the SRF, or from another cluster via the inter-cluster switch. This organization is borrowed from Imagine. We expect that more efficient register organizations exist and we are currently evaluating alternative organizations. We are also revisiting the choice of 64 registers per LRF (768 registers per cluster). We expect that we will be able to
reduce the register count significantly as the compiler register allocation algorithms improve.

At the top-level of the Merrimac node floorplan, the clusters are organized in an array to minimize the latency and power for inter-cluster communication. This cluster array occupies the right 60% of the Merrimac die. In the center of the array is the microcontroller which provides the clusters with VLIW instructions from its internal microcode store. The small gaps between the clusters contains the inter-cluster switch which can exchange one word per cluster per clock cycle.

The left 40% of the Merrimac processor chip contains the two scalar processors, the memory system, and the network interface.

Two scalar CPUs are used to make the scalar processor self-checking. One processor is the master. The other operates as a shadow processor (receiving the same inputs but generating no outputs). The two processor’s results (and many internal signals) are compared each cycle. Any mismatch on the comparison triggers an error recovery. To achieve adequate reliability in large configurations self checking operation is required or the rate of undetected soft errors in the scalar processor will be unacceptably high.

To estimate the area and power for each of the two scalar CPUs we use data from the most complex synthesizable 64 bits processor core available to us: MIPS64 20kc.

The stream memory system connects the SRF to external memory. Address generators fill or empty the SRF’s data by issuing memory addresses to the 8 cache banks through the memory switch. Memory addresses and data go through the network interface or the appropriate RDRAM controller.

The memory system reorders memory accesses to take advantage of higher throughput on certain access patterns. Reads returning from memory are then reordered into their original request order in the reorder buffers. On memory reads, the data gets corrected by a SECDED\(^6\) code.

The overall area of the Merrimac processor chip is 10.2mm $\times$ 12.5mm ($128\text{mm}^2$). This is a very small die compared to conventional microprocessors that range from 200 to over 300mm\(^2\) in area. With this small die area, we expect to be able to produce Merrimac processor chips in quantity for less than $200 each. This small die area also makes it possible to expand the architecture. We

---

\(^6\)Single error correcting, double error detecting.
have adequate room to add clusters, add arithmetic units within the clusters, increase the size of the memory arrays, and/or add other features, and still hit our cost targets.

<table>
<thead>
<tr>
<th>Unit</th>
<th>Count</th>
<th>Power(each) (W)</th>
<th>Power(total) (W)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalar CPU</td>
<td>2</td>
<td>1.10</td>
<td>2.20</td>
</tr>
<tr>
<td>Microcontroller</td>
<td>1</td>
<td>0.60</td>
<td>0.60</td>
</tr>
<tr>
<td>MADD ALU</td>
<td>4/cluster</td>
<td>0.11</td>
<td>7.04</td>
</tr>
<tr>
<td>Local Register Files (per MADD ALU)</td>
<td>4/cluster</td>
<td>0.02</td>
<td>1.28</td>
</tr>
<tr>
<td>SRF bank</td>
<td>1/cluster</td>
<td>0.09</td>
<td>1.44</td>
</tr>
<tr>
<td>Intra-cluster switch</td>
<td>1/cluster</td>
<td>0.04</td>
<td>0.64</td>
</tr>
<tr>
<td>Cluster</td>
<td>16</td>
<td>0.67</td>
<td></td>
</tr>
<tr>
<td>Inter-cluster switch</td>
<td>1</td>
<td>0.69</td>
<td>0.69</td>
</tr>
<tr>
<td>Cache bank</td>
<td>8</td>
<td>0.12</td>
<td>0.96</td>
</tr>
<tr>
<td>Memory controller</td>
<td>1</td>
<td>5.00</td>
<td>5.00</td>
</tr>
<tr>
<td>Network controller</td>
<td>1</td>
<td>5.00</td>
<td>5.00</td>
</tr>
<tr>
<td><strong>Total Merrimac Chip</strong></td>
<td></td>
<td></td>
<td><strong>25.09 W</strong></td>
</tr>
<tr>
<td>DRAM chip</td>
<td>16/Merrimac Node</td>
<td>1.00</td>
<td>16.00</td>
</tr>
<tr>
<td><strong>Total Merrimac Node</strong></td>
<td></td>
<td></td>
<td><strong>41.09 W</strong></td>
</tr>
</tbody>
</table>

Table 4.2: Power Breakdown

The power estimates for a Merrimac node chip are broken down in table 4.2. Again these numbers are based on a 90nm technology with a supply voltage of 1.0V and a clock frequency of 1GHz. At 26W, the chip can be cooled with a simple heat sink and conventional forced-air convection cooling. The total node budget of 41W gives a 16-node board power of roughly, 650W which is a concern and may require a more expensive cooling system.

4.1.2 Indexed SRF

The stream register file (SRF) is a key component of the storage hierarchy of a stream processor. It captures producer-consumer locality between kernels, and provides latency tolerance by allowing transfers to and from memory to be overlapped with computation on data already in the SRF. (This latency hiding is similar to that provided by a vector register file on a vector computer such as the X-1). Conventional stream or vector register file implementations restrict accesses to streams or vectors to be sequential (or some small number of predetermined access patterns). However, several classes of applications within the domain of scientific computing, as well as in other fields such as signal processing and cryptography, exhibit data reuse patterns that are not amenable to sequential accesses. Therefore, the sequential access restriction of the SRF presents an artificial impediment to extending the stream programming model to such applications that would otherwise be promising candidates.

In order to efficiently support this wider class of applications, the Merrimac architecture supports explicitly indexed, arbitrary order access to data in the SRF as well as sequential stream accesses. The following subsections provide a qualitative discussion of the benefits of indexed SRF access, examples of application constructs that benefit from indexed accesses, and an evaluation of the concept using a set of benchmarks from target application classes.

**Benefits of Indexed SRF access**

Temporal locality available in applications at the stream level can be broadly categorized based on the ordering of reused data as follows:
4.1. MERRIMAC ARCHITECTURE

**In-order stream locality:** Multiple accesses to a stream in the same order by one or more kernels.

**Reordered stream locality:** Multiple accesses to data within a stream in different order(s) by one or more kernels.

**Intra-stream locality:** Repeated access to subsets of data elements within a stream during a single kernel execution.

Strictly sequentially accessed SRFs only exploit in-order stream locality. The key performance benefits of indexed SRF access come from three main sources:

Capture more temporal locality: Unlike a sequentially accessed SRF, which requires writing the data out to memory and re-reading in the desired order, indexed SRF access allows reordered stream locality to be captured in the SRF. This results in a reduction in memory bandwidth and a resulting performance improvement for memory-limited applications.

Reduced data replication: Repeated access to the same data within a stream requires that data be replicated in a sequential SRF. Indexed access, on the other hand, allows repeated reference to a single copy of the data, reducing capacity pressure in the SRF. Most applications have at least some data sets that are too large to fit in the SRF. These data sets are partitioned into multiple *strips*. With indexed SRF access, eliminating replication results in a smaller number of larger strips, reducing associated overheads. There is also a reduction in memory traffic as redundant reads from memory are eliminated.

Efficient support for conditional and complex accesses: SRF index computation provides a powerful technique for expressing conditional and complex access patterns at a very fine granularity. In addition, it is sometimes beneficial to structure memory accesses to optimize DRAM throughput and perform complex permutations via SRF indexing.

**Application Constructs**

Indexed SRF access enables efficient support for several constructs commonly used in classes of applications that are amenable to stream programming. A few of these are summarized below.

Multi-dimensional array accesses: Accesses along different dimensions of a multi-dimensional data array can be supported efficiently by capturing reordered stream locality in the SRF.

Neighbor accesses in multi-dimensional arrays: Accessing data points neighboring a given element in a multi-dimensional array with sequential streams requires reordering to place adjacent values in all dimensions contiguously within the stream. In addition, avoiding data replication without SRF indexing requires complex state management in the kernels using local scratch-pad memories. Indexed SRF access makes it trivial to perform the neighbor accesses directly from a single copy of the data in the SRF.

Table lookups: Data-dependent table lookups are used in a wide variety of applications both due to algorithmic requirements and as an optimization to store pre-computed values. With indexed SRF access, lookups can be performed from the SRF if the table or a useful partitioning of it can be accommodated in the SRF.

Neighbor accesses in irregular graphs: Accessing neighbors of nodes in irregular graph structures using sequential streams result in nodes that neighbor multiple other nodes being replicated. The intra-stream locality of these accesses is captured easily in an indexed SRF.

Conditional accesses: Conditional control flow is an expensive operation in SIMD processors [11], [24]. In some cases, conditional SRF index computation provides a powerful and convenient technique for expressing conditionals with low overheads.
CHAPTER 4. MERRIMAC: THE STREAMING SUPERCOMPUTER

Evaluation

This section presents the performance of several applications and synthetic benchmarks with and without indexed SRF access. Please note that these results are from a general study on indexed SRF access that was performed on an 8-cluster machine. However, the compute and communication resources as well as register and memory bandwidths are identical to Merrimac when scaled to 16 clusters.

We consider two levels of indexing freedom. In-lane indexing allows a compute cluster to index into its local bank of the SRF only. Cross-lane indexing allows any cluster to index into any bank of the SRF. Estimates of SRF implementation overheads show that in-lane and cross-lane indexing can be supported with 11% and 22% increases in area over a sequentially accessed SRF respectively. This increase in SRF area corresponds to a 1.5% to 3% increase in overall die area based on data extrapolated from the Imagine processor implementation [15]. Note that the area estimates presented in Subsection 4.1.1.

<table>
<thead>
<tr>
<th>Base</th>
<th>Sequential SRF backed by off-chip DRAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISRF1</td>
<td>Indexed SRF with one word per cycle per lane in-lane and cross-lane indexed bandwidth backed by off-chip DRAM</td>
</tr>
<tr>
<td>ISRF4</td>
<td>Indexed SRF with up to 4 words per cycle in-lane and 1 word per cycle cross-lane indexed bandwidth backed by off-chip DRAM</td>
</tr>
<tr>
<td>Cache</td>
<td>Sequential SRF backed by on-chip cache and off-chip DRAM</td>
</tr>
</tbody>
</table>

Table 4.3: Machine configurations simulated

<table>
<thead>
<tr>
<th>2D FFT</th>
<th>2-dimensional FFT on 64x64 data array (corner-turn after 1-D computation)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rijndael</td>
<td>Block cipher with large number of table lookups</td>
</tr>
<tr>
<td>Sort</td>
<td>Merge-sort of 4096 values</td>
</tr>
<tr>
<td>Filter</td>
<td>5x5 convolution filter on 256x256 2-D array</td>
</tr>
<tr>
<td>IG_SML</td>
<td>Neighbor access in irregular graph (avg. graph degree 4, FP ops per neighbor 16)</td>
</tr>
<tr>
<td>IG_SCL</td>
<td>Neighbor access in irregular graph (avg. graph degree 4, FP ops per neighbor 51)</td>
</tr>
<tr>
<td>IG_DMS</td>
<td>Neighbor access in irregular graph (avg. graph degree 16, FP ops per neighbor 16)</td>
</tr>
<tr>
<td>IG_DCS</td>
<td>Neighbor access in irregular graph (avg. graph degree 16, FP ops per neighbor 51)</td>
</tr>
</tbody>
</table>

Table 4.4: Benchmark descriptions

The performance of benchmarks is evaluated for four machine configurations. Table 4.3 describes the machine configurations, and Table 4.4 briefly describes the benchmarks. All simulations were performed using a cycle accurate simulator. Benchmark kernels were written in KernelC and scheduled using an automated scheduler based on [21].

Figure 4.4: Off-chip memory traffic normalized to Base
4.1. MERRIMAC ARCHITECTURE

Figure 4.4 shows the off-chip memory bandwidth requirements of the benchmarks for ISRF and Cache configurations, normalized to the Base case (note that ISRF1 and ISRF4 have identical memory bandwidth requirements). Indexed SRF access provides significant bandwidth savings for all benchmarks except Sort and Filter, which have no reordered or intra-stream reuse. 2D FFT, Rijndael, and all IG benchmarks benefit from capturing reordered or intra-stream locality in the SRF. The Cache configuration also reduced bandwidth as much as the ISRF did for statically analyzable benchmarks (2D FFT and Rijndael), and captured even more locality than ISRF for irregular, data-dependent benchmarks by exploiting partial inter-stream locality as well.

We expect a combination of indexed SRF and cache, as is the case in Merrimac, to reduce bandwidth slightly from the cache-only case for the IG benchmarks by reducing conflict misses in the cache, partially offset by the overhead of an additional index stream required to index into the SRF. The current IG data sets, however, are too small to evaluate this effect, and we are currently exploring this issue using larger data sets. The other benchmarks in figure 4.4 do not benefit further beyond the indexed-SRF-only or cache-only case in terms of bandwidth savings.

![Figure 4.5: Execution times normalized to Base. ISRF1 shown only where it differs from ISRF4.](image)

Figure 4.5 shows the breakdown of execution time for all four configurations. ISRF1 and ISRF4 differ only for benchmarks with more than one indexed stream (in our benchmark set, only Rijndael and Filter have this requirement).

ISRF4 provides speedups over the Base configuration for all benchmarks. FFT 2D and Rijndael on the Base machine are constrained by memory bandwidth, and reducing memory traffic via indexed SRF access provides speedups of 2.24x and 4.11x respectively.

Improvements in the Sort benchmark on ISRF4 are due to efficient conditional SRF access, resulting in reduced kernel loop execution time. The speedup of the Filter benchmark is due to efficient access of neighbor values directly from the SRF, also reducing kernel loop execution time.

The IG benchmarks span a wide range of application characteristics. IG_SML and IG_DMS have low compute density and are constrained by memory bandwidth. Capturing intra-stream locality of these data sets in the SRF improves performance of both. Another factor contributing to improved performance on ISRF4 is the increased strip sizes that can be accommodated in the SRF as a result of eliminating replication, which amortizes kernel start and end overheads over larger batches of useful computation. The two dominant components of these overheads for this benchmark are software pipeline overhead and load imbalance between lanes. While we do implement dynamic load balancing using the technique presented in [11], some imbalance still exists at the end of each strip as all lanes remain occupied until the last lane has completed processing its final input. These overheads represent a more significant portion of the run time for IG_DMS and IG_DCS which have
shorter strip sizes. Other overheads such as application initialization operations are minimal in this benchmark, but may be significant in real applications, which would further benefit ISRF4.

IG_SCL represents a scenario where SRF indexing provides very little performance benefit since it is compute-limited and has long strips even on the Base configuration.

ISRF4 also outperforms the Cache configuration for most benchmarks. The only exception is IG_SML where the added locality captured in the cache provides an advantage as that benchmark is memory limited on all machine configurations. FFT 2D data set reordering is fully captured in the cache, but it is not eliminated as in the case of ISRF4. Therefore, the software pipeline loop in the Cache configuration is longer than in ISRF4, limiting the performance benefit from the cache as the iteration interval is bounded by the number of data sets that fit in the SRF simultaneously. For Rijndael, the cache fully captures the table lookups, but does not have adequate bandwidth to eliminate all memory stalls. The cache does not provide the conditional and complex SRF accesses enabled by ISRF4 that benefit Sort and Filter, and does not provide any speedup for these benchmarks. The cache also does not eliminate data replication in the SRF, and therefore does not provide the strip size improvements that benefit IG_DMS and IG_DCS on ISRF4.

Finally, none of the benchmarks suffer significantly from a lack of indexed SRF bandwidth on ISRF4. Rijndael and Filter spend 42% and 18% of execution time on SRF stalls on ISRF1 demonstrating the benefit of high indexed SRF bandwidth for some benchmarks.

Conclusions

Indexed SRF access provides significant performance improvements for applications that require non-sequential stream reuse or complex/conditional access patterns. In-lane indexed access largely benefits applications with access reorderings that can be statically analyzed, and data placement orchestrated at compile-time. In-lane indexing is also sufficient for much of the complex and conditional access patterns, and incurs very little overhead in terms of area and design complexity. On the other hand, cross-lane indexing further extends the class of applications that can be efficiently streamed by providing support for irregular, data-dependent access patterns, but at an increased area and complexity cost.

Since the Merrimac architecture contains a cache that provides much of the additional benefits provided by cross-lane indexing, the current design proposal calls for in-lane indexed SRF access only. Cross-lane indexed access is fully supported in the tool chain (compiler and cycle-accurate simulator), and is being investigated further as the set of application under investigation broadens and new requirements emerge.

For a more detailed treatment of the issues discussed in this section, as well as sensitivity analyses and a description of the hardware implementation, please refer to [10].

4.1.3 Fused Multiply-Add

The multiply-add operation is important in many scientific and engineering applications. Recently, the floating-point unit of several commercial processors include a floating-point multiply-add-fused (MADD) unit, which executes the double-precision multiply-add $f = A + (BC)$ as a single instruction. This fused implementation has two main advantages:

- The operation combines two basic floating-point operations in one, with only one rounding error as opposed to two. This tends to reduce overall rounding errors. In common cases this difference may be relatively unimportant, but in special situations, the lack of an intermediate rounding makes possible a number of techniques that are difficult or costly on a traditional architecture.
There is reduction in the delay and hardware required by sharing several components. In typical implementations, the final addition can be combined into the floating-point multiplication operation without significantly increasing its latency. Thus, a single fused multiply-add is faster than a multiplication and an addition executed successively.

As described in detail in Subsection 4.1.1, each Merrimac compute cluster includes four multiply-add functional units. In this section we will describe several algorithms that have been incorporated into our kernel scheduler, and automatically detect opportunities for fusing operations.

Fusing floating-point operations

In a VLIW architecture, the operations to be executed should be known before scheduling them on to the different functional units available. In order to take advantage of the MADD units, potential floating-point multiply and add/sub operations need to be fused as a pre-scheduling step, as shown in Figure 4.1.3 and in the example in Figure 4.7. Note that fusing replaces the add/sub operations with the fused add/sub operation. The multiply operation can be removed if there are no remaining operations dependent on it. Thus, the number of operations either remains constant or decreases.

**Greedy Approach** (*Greedy*)

The most obvious approach is to have a graph traversing algorithm that traverses the data-flow graph and on incurring a floating-point multiply operation, fuses all successor floating-point add/sub operations together. Note that there are two obvious drawbacks to this scheme.

Our main objective is to get the best schedule possible, i.e the shortest schedule possible. This requires that the most critical path be reduced to its full potential. By fusing operations in a greedy way, we may be losing many opportunities for fusing along the critical path. An example of this is shown in Figure 4.8 where both operands of an addition are both results of a multiply operation. The greedy algorithm will fuse the first multiply it comes across which is not necessarily on the critical path.

The second problem is that depending on the order in which operations are traversed we may miss fusing opportunities. For example, consider the case where a multiply operation produces a result that is consumed by several addition operations whose second operand is also a result of a multiplication. If the first multiply is fused with all of its dependent additions no other multiply can be fused, leading to many lost opportunities (Figure 4.9).
Note that this is a very simple scheme and if the order in which the operations in the graph are traversed is picked properly, we can achieve good schedules. For our experiments, we perform a breadth-first traversal from top to bottom of each basic block. The results presented later show that this is a very effective scheme when used on the compute-intensive test kernels.

**Critical Path Sensitive Approach (Crit)**

This scheme fixes the first problem. Instead of traversing in random order and exploring each operation and all its children (those who feed on the result produced by the operation), it traverses the graph in the critical path order. This can be done by putting weights on each of the edges in the data-flow graph, denoting their criticality. Criticality is a measure of the latency from the current operation to the leaf operation (last operation in the basic block). On encountering any possibility of fusing as in Figure 4.10, we fuse the operations on the critical path and then recalculate the weights since the latencies have changed. Since we want to go over all the operations, once we reach a leaf, we traverse the next most critical path and go on doing so until all paths are traversed.

This scheme has the overhead of recalculating the weights after every fusing opportunity has been explored. This adds to the run-time complexity, which in the worst case will be in the order of half the edges in the data-flow graph. Moreover, it traverses each path as opposed to each node.

Figure 4.7: Example - Multiply Operation Feeding Many Addition Operations
Initiation Interval Approaches

Performance of software pipelined blocks is dependent on both the initiation interval (II) as well as the block length. The II is the interval between consecutive initiations of the software pipelined loop. The run time of a kernel is determined by the II:

\[
\text{run time} = \text{number of iterations} \times \text{II} + \text{block length}
\]  

(4.1)

Reducing the II improves the performance for each iteration and hence is more critical than the block length. The II is determined by both the SWP dependency paths and resource constraints [17].

SWP Dependencies (RecurII)

If the II is determined by the SWP dependencies, it is most beneficial to reduce the length of the dependency path. Thus, the first step for a software-pipelined block is to fuse as many operations as possible along the II critical path. This can be done by traversing only this path just like the most critical path in the Crit approach. Once the II critical path is shortened to its minimum, the II determined by dependencies in the data-flow graph cannot be reduced any further. Hence, we follow it with the Crit approach, resulting in a scheme that is II and critical-path sensitive.

Resource Constraints (RedII and CritII)

Let us now explore the other factor which determines the II: the number of functional units available. To explain this, let us first divide a basic block into \( n \) stages, where each stage is as long as the II. At any given cycle, operations from different iterations of the loop can be overlapped (software-pipelined), provided they belong to different stages. Thus, the number \( n \) is also the number of iterations that can be overlapped at any given cycle. However, a valid schedule must also conform to the resource constraints of the architecture, we must able to schedule all operations in a single cycle on the limited number of functional units. Thus, we cannot always achieve the minimal II set by SWP dependencies prolonging the II and hence the schedule. Since we are fusing operations during a pre-scheduling step, we cannot determine on which functional unit any given operation will be scheduled. We will assume the best case scenario where all operations are uniformly distributed.
over the functional units they can be scheduled on. This gives us the best possible II due to resource constraints. Now, in order to decrease this II, we must decrease the number of operations contending for the same functional unit. Since the most important unit in our VLIW cluster is the MADD unit, we should cut down the number of multiply operations by simply fusing as many of them as possible.

This leads us to a new scheme, called RedII. Note that we still have the problem illustrated in Figure 4.8, i.e both of an add/sub operation operands are generated by multiply operations and we must choose which one to fuse. If one of the multiply operations feeds its result only to add/sub we will be able remove the multiply operation once it is fused with all its dependents, thus achieving our goal. When both the multiply operations exhibit the same credentials we choose using the critical path criterion.

Since one of our goals is to minimize the total number of operations, all multiply operations whose results are used by a single add/sub operation should be fused allowing us to remove the multiply operation. Thus, in cases like Figure 4.8, the very first thing we need to check is whether the other multiply operation(m1) has only one successor in the form of an add/sub operation. If so, clearly the other other multiply operation is fused. Note that this sacrifices reduction in the critical path, which is not our primary goal. Next we need to find out if the current multiply operation itself can be removed, i.e if all its successors are add/sub operations and hence can be fused. Again, there can be other multiply operations generating results for the same add/sub operations and can themselves be removable, i.e have all fusible successors. An interesting scenario occurs when the other multiply operation(s) are also in the critical path. Note that we have two fusing opportunities here, thus reducing the critical path twice. However, if any one of them fuses all its successors and
removes the multiply operation, we will be able to reduce the critical path only once. Moreover, note that reducing a single operation does not necessarily reduce the II, i.e. we need at least as many operations as the MADD units to reduce the II by one (pertaining to our assumption of uniform distribution). This introduces a tradeoff and our new scheme, \( \text{CritII} \), opts to reduce the critical path twice, while \( \text{RedII} \) removes the current multiply operation, thus losing the other fusing opportunity and critical path reduction.

**Results**

The five different algorithms were implemented in the kernel scheduler. We studied their respective performance on different kernels of two of our applications: streamMD and streamFEM-2D. The streamMD kernels are:

1. Compute\_force\_SP
2. Compute\_force\_SP\_strip

The streamFEM-2D kernels are a function of the polynomial order, and the 1-D and 2-D quadrature points, each increasing the kernel size and complexity monotonically. Three different point sets were tried: \((0, 1, 1), (2, 3, 7)\) and \((3, 4, 16)\). The 2-D streamFEM kernels are:

1. GC\_nq
2. GFI\_nq
3. GFB\_nq
4. GFAC\_nq
5. GFACL\_ne\_nq

For the third point set in streamFEM-2D, the scheduler faced register allocation problems with the GC and GFAC kernels. All the above-mentioned kernels were software pipelined and their IIs were determined by resource constraints due to the sheer volume of instruction-level parallelism (ILP) available. The VLIW cluster currently has only 4 MADD units and other configurations have not been explored. The following sub-sections analyze the different performance metrics for each of the 5 fusing algorithms.
Multiply Operations Reduced

Figure 4.11 gives us an indication of how efficient the algorithms are in terms of reducing the number of operations and hence the II. As expected, RedII is a clear winner here, and its impact is most pronounced in the GFB kernels. Also note that CritII performs better than Crit since all the kernels are software-pipelined and hence II limited. However, the decisions made in favor of critical path degrade the performance of CritII as compared to RedII. Another interesting result to note is that Greedy performs better than the other schemes, except RedII, for kernels like GFB and GFI. This is a clear indication of the weakness in the critical-path sensitive approaches for software-pipelined blocks.

![Percentage of Multiplier ops removed](image)

Figure 4.11: Percentage of Multiply Operations Removed

Figure 4.12 takes a closer look at the results of RedII, which achieves close to 40% reduction in number of operations on average.

![Percentage FMUL removed by RedII](image)

Figure 4.12: Percentage of Multiply Operations Removed by RedII

Addition Operations fused

It is not surprising to note that the number of addition operations does not change with the different schemes because all the schemes stop fusing when there are no more add/sub operations to be fused. It is only the choice of multiply operation with which they are fused that changes
between the different algorithms. So, we only present the results for RedII in Figure 4.13. It is interesting to note that close to 80% of the add/sub operations were involved in fusing, and the full 100% for the GC and GFAC kernels. Most importantly, this result verifies the correctness of the different schemes in exploring every possible fusing opportunity.

![Percentage of Adder ops fused using redII](image)

**Figure 4.13: Percentage of Additions Fused by RedII**

**Improvement in II**

![Percentage Improvement in II](image)

**Figure 4.14: Percentage improvement in Achieved II**

This is the main performance metric since we want the best possible schedule for each of our kernels. Note that this is the II achieved by the scheduler, and not the minimum II possible due to dependencies alone. We explore this first and then we take a look at the improvement in the best minimal II for each of these schemes. Since it has already been established that the IIIs for each of the kernels studied is determined by the resource constraints and the maximum reduction in multiply operations is achieved by RedII, it is no surprise to find it the best performer in terms
of II improvement as well (Figure 4.14). The effect is again most pronounced in the GFB kernels. While close to 20% reduction is registered on average, around 50% reduction is achieved in a couple of cases. These results seem to be in conflict with the reduction in multiply operations. On further investigation, we concluded that the new schedules are now limited by other functional units, namely the scratch-pad unit. This is quite evident in the case of GFI.

Conclusions and Future Work

It is clear from the above results that for software-pipelined blocks, it is critical to have schemes which can reduce the II (Initiation Interval) as opposed to just the critical path. For kernels limited by the available number of resources, RedII, which tries to remove as many multiply operations as possible, is the best algorithm tested.

However, when the II is limited by the SWP dependencies, we should first try to reduce the II critical path. RedII and CritII are devised to achieve this, but the effect was not visible in any of the scientific kernels since all of them have very large amounts of ILP. It is expected that CritII will perform better when the II is only limited by SWP dependencies since it also tries to minimize the critical path (basic block length), unlike RedII. Note that it is easy to automatically choose which of the two schemes to apply, as we first estimate the II and can determine the limiting factor.

For non-software pipelined blocks, both Crit and CritII will perform better than the other techniques since they minimize the critical path and which determines the basic block length. Since CritII is more aggressive than Crit in terms of reducing the number of operations it is expected to reduce the load on the functional units more, and hence may result in a better schedule. On the other hand, Crit fuses operations in critical path order and hence may perform better when there are more than one critical paths closely interacting. These effects need to be analyzed further by exploring more kernels. However, since kernels for scientific applications are by nature compute-intensive and hence provide numerous opportunities for ILP we do not expect these experiments to yield positive results.

Greedy is a much simpler scheme with smaller run-time costs and is effective when the achieved IIs are compared. Though the top-to-bottom ordering of visiting each operation contributes to this behavior, the randomness in breadth-first traversal can contribute to bad results in some cases. However, for compute-intensive kernels like the ones explored, there are many opportunities of fusing and Greedy has a high probability of taking advantage of most of them.

The extra register file added for the MADD unit is not fully utilized by our current scheduler. This register file can also be used to store the conditional code (CC) variables, currently stored in the CCRF. Not only will this increase the occupancy of the extra register file, it will also be helpful in removing the global connections between the CCRF and each of the MADD units.

One issue that remains to be explored is the tradeoffs regarding the higher power consumption of the fused multiply-add functional units.

4.1.4 Iterative Operations

The sixteen clusters of the Merrimac stream processor exploit instruction-level parallelism (ILP) using a very-long instruction word (VLIW). One issue with such an organization is to determine the mix of functional units. The Imagine stream processor, for example uses clusters containing three floating-point adders, two floating-point multipliers, and one floating-point divider/square-root unit. While this $3+2	imes,1\div/\sqrt{\cdot}$ worked well for the signal processing applications for which Imagine was designed we found that it often led to load imbalance on the scientific applications for which Merrimac is targeted. The StreamMD application (Section 4.4.4) for example, requires many divides and square roots fully subscribing the divide/square root unit while leaving the multipliers idle. Other applications use primarily multiplies and adds, leaving the divide/square root unit
idle. In addition to leading to such load imbalance on different programs, dedicated divide/square root units are also very costly to implement in terms of die area — particularly if they are fully pipelined.

To solve this load balance problem while keeping a small cluster area, we have studied an alternative implementation for Merrimac that uses four identical FPUs each of which can perform a multiply-add, \( A \times B + C \) (or subtract, \( A \times B - C \)) operation. These four identical units perform all arithmetic functions. Multiplies and adds are performed directly (possibly fused) while reciprocals, square-roots, and a number of other elementary functions are performed as a sequence of table lookups, multiplies, and adds. There are currently 2 methods being evaluated to perform division and elementary operations, the Newton-Raphson method and Taylor series expansion.

**Newton-Raphson**

The Newton-Raphson Method works by solving the equation \( f(X) = 0 \), for any function \( f \), using gradient search. The Algorithm starts with an initial approximation of the root \( X_0 \). The next approximation to the root, \( X_1 \), is found by projecting the tangent of the function \( f(X) \) at \( X_0 \), to the x-axis. Each iteration reduces the error until the root has been found to a desired precision.

\[
X_{i+1} = X_i - \frac{f(X)}{f'(X)}
\]

for division, we find \( \frac{1}{b} \) by setting \( X = \frac{1}{b} \), \( f(X) = \frac{1}{X} - b \) then \( f'(X) = -\frac{1}{X^2} \)

\[
X_{i+1} = X_i - \frac{\frac{1}{X_i} - b}{-\frac{1}{X_i^2}} = X_i(2 - bX_i)
\]

This error has been proven to converge quadratically (\( E_{i+1} = |bE_i^2| \)) in [9]. Using a MADD unit, each iteration takes 1 MUL-ADD and 1 MUL. Hence the latency to get \( \frac{1}{b} \) given more than one unit (compute \( a \times X_i \) in parallel with \( (2 - bX_i) \) for the last iteration) is for \( L \) iterations:

\[
t_{\text{divNR}} = t_{\text{lookuptable}} + L(t_{\text{madd}} + t_{\text{mul}})
\]

It occupies \( 2L + 1 \) multiply slots.

Similarly for \( X = \frac{1}{\sqrt{b}} \), we set \( f(X) = X^{-2} - b \), \( f'(X) = -2X^{-3} \)

\[
X_{i+1} = X_i - \frac{X^{-2} - b}{-2X^{-3}} = \frac{1}{2} X_i(3 - X_i^3b)
\]

Hence each iteration requires 4 operations (unless there is an automatic divide by 2, then 3).

\[
t_{\text{sqrtNR}} = t_{\text{lookuptable}} + 4Lt_{\text{mul}}
\]

It occupies \( 4L + 1 \) multiply slots.

**Taylor Series Expansion**

This comes from Albert Liddicoat’s PhD thesis [19]. Basically we start with the \( n^{th} \) order Taylor polynomial approximation of function \( f \) at 0.

\[
p_n(x) = f(0) + f'(0)x + f''(0)\frac{x^2}{2!} + ... + f^{(n)}(0)\frac{x^n}{n!}
\]

Remembering that for division we use \( b = 1 + x \), here are basic functions’ Taylor series expansions:
\[
\frac{1}{1 + X} = 1 - X + X^2 - X^3 + X^4 - \ldots
\]
\[
\sqrt{x} = 1 + \frac{x}{2} - \frac{x^2}{8} + \frac{x^3}{16} - \frac{15x^4}{128} + \ldots
\]
\[
\frac{1}{\sqrt{x}} = 1 - \frac{x}{2} + \frac{3x^2}{8} - \frac{5x^3}{16} + \frac{35x^4}{128} - \ldots
\]

The basic form is thus
\[
f(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + a_4 x^4 + \ldots
\]

To accelerate convergence, \(x\) can be pre-scaled, then the series computation can be post-scaled by multiplying the computed series by the appropriate post-scaling constant.

\[
\frac{a}{b} = aX_0(1 + d + d^2 + d^3 + d^4)
\]
\[
\sqrt{b} = Y_0(1 - \frac{1}{2}d - \frac{1}{8}d^2 - \frac{1}{16}d^3 - \frac{15}{128}d^4)
\]
\[
\frac{1}{\sqrt{b}} = Z_0(1 + \frac{1}{2}d + \frac{3}{8}d^2 + \frac{5}{16}d^3 + \frac{35}{128}d^4)
\]
\[
d = (1 - bX_0)
\]
\[
X_0 \approx \frac{1}{b}
\]
\[
Y_0 \approx \sqrt{b}
\]
\[
Z_0 \approx \frac{1}{\sqrt{b}}
\]

In the divide case, it takes 2 parallel MADD to evaluate \(d\) and \(aX_0\), another MUL for \(X\).

The error for division is \(E_q = abE_{lookuptable}^{k+1}\) for the \(k\)th order approximation. The \(k\)th order approximation multiplies the bits of precision of \(X_0\) by \((k + 1)\).

Now the convergence of the error is not as fast as Newton Raphson, but it can be done in parallel and with much simpler units. Because we are elevating numbers of limited precision to powers (or multiplying them by simple coefficient, this can be done very effectively by simple logic as shown in [19].

**Proposed Alternatives**

We consider two alternative implementations.

**Simple:** At the minimal case, we could have a 8/9 bits lookup table for an approximation to \(\frac{1}{\sqrt{x}}\).

To get an approximation for division we would have to square the look-up value. We would then need 3 iterations to get 64 bits of precision for either reciprocal or square root.

**Accelerated:** An 11-bit table is used for \(\frac{1}{x^2}\), and a separate 8-bit table for \(\frac{1}{\sqrt{x}}\). This method would require 5 products of the Taylor Series expansion method. The multiplication of \(a \times X_0\) and the \((1 - bX_0)\) would be done on MADD units as well as the final multiplication. A special ALU would compute the polynomial product.
4.1. MERRIMAC ARCHITECTURE

Here is a summary of characteristics for the implementations where N is the number of MADD units per cluster

<table>
<thead>
<tr>
<th>Solution</th>
<th>Operation</th>
<th>MADD</th>
<th>Latency</th>
<th>Sustained throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple</td>
<td>Division</td>
<td>8</td>
<td>$t_{LU} + 7t_{MADD}$</td>
<td>N/8</td>
</tr>
<tr>
<td></td>
<td>Inverse Sqrt</td>
<td>10</td>
<td>$t_{LU} + 9t_{MADD}$</td>
<td>N/10</td>
</tr>
<tr>
<td>Accelerated</td>
<td>Division</td>
<td>3</td>
<td>$t_{LU} + 2t_{MADD} + t_{spec}$</td>
<td>N/3</td>
</tr>
<tr>
<td></td>
<td>Inverse Sqrt</td>
<td>3</td>
<td>$t_{LU} + 2t_{MADD} + t_{spec}$</td>
<td>N/3</td>
</tr>
</tbody>
</table>

The plan of record is to use the indexable SRF to store the approximates and then use the Newton-Raphson method on the Multiply add units. This requires only a special addressing mode to use the most significant bits as the index for the lookup. We have yet to evaluate the area cost of a specialized ALU unit and see if it makes sense to use it to speed up division and other elementary functions.

4.1.5 Stream Memory Architecture

The Merrimac memory system handles memory requests such as stream loads and stream stores by receiving streams from or sending streams to the stream register file. Because the memory system treats streams instead of individual words, it is optimized for throughput rather than latency. Memory latency is hidden by the parallelism of stream operations and thus is not an issue. Each Merrimac processor chip includes 16 DRAM interfaces that together provide a peak off-chip memory bandwidth of 38.4GB/s. This is nearly an order of magnitude more local memory bandwidth than peak bandwidth typical high-end microprocessors offer on their front-side buses (3-6GB/s). Random access bandwidth is considerably lower than peak bandwidth in both cases, about 16GB/s for Merrimac.

In addition to high DRAM bandwidth, the Merrimac memory system includes a stream cache that acts as a bandwidth multiplier for streams with re-use and an add & store unit that supports remote memory operations. The stream cache consists of eight cache banks that are interleaved on the low bits of the 2-word cache line address. Each cache bank has a throughput of one word per cycle for an aggregate cache bandwidth of 64GB/s. Because a stream processor is not sensitive to latency, the purpose of the stream cache is entirely bandwidth amplification. This is in contrast to a conventional processor cache which primarily functions to reduce latency. For applications with re-use, the stream cache boosts memory bandwidth by up to a factor of four. To effectively use the
stream cache, it must be managed by software, both to ensure coherence and to avoid polluting the cache with data that is re-used through the SRF. Compiler algorithms for managing the cache are a subject of current study.

The stream add & store unit is designed to process remote memory operations (add & store and fetch & add operations). Add & store is an atomic operation that adds its operand to the contents of the effective memory address \((M \leftarrow M + D)\) rather than replacing the contents of the effective memory address with the operand as in a write \((M \leftarrow D)\). Fetch & add is an atomic operation that returns the result \((M + D)\) of an add & store operation. The address generators expand a single stream add & store or fetch & add operation into an array of scalar operations. Each scalar remote memory operation is then handled by the memory system. A stream add & store operation is also called a scatter-add or scatter-op operation when the target addresses of the operation is spread over memory space using an indexed stream addressing mode. Remote operations are performed by an integer and floating-point add unit in each cache bank as shown in Figure 4.15.

The fetch & add primitive is particularly useful when adding or removing elements from shared queues. A fetch & add to the queue pointer atomically increments the pointer and returns the address at which the next element should be read or written. With this approach multiple nodes can simultaneously add or remove elements from the same queue without needing to first acquire a lock.

The store & add primitive is useful for operations like computing a histogram of a set of values. Each element of the set simply performs an add & store on the memory location holding the appropriate histogram bin. As with fetch & add, multiple nodes can increment this bin simultaneously. The memory system combines references to the same bin.

Privatization and sort&count algorithms have been introduced for computing histograms on vector architectures, and these algorithms can be used on a stream architecture without store & add. However, these privatization algorithms do not perform as well as directly computing the histogram using store & add. The privatization algorithm counts the number of elements whose values fall in each bin and repeats this process over the range of values elements can have. In this algorithm the execution time is proportional to both \(N\) and the range of element values. We can get the histogram of a stream with the time complexity of \(O(N)\) if the stream is sorted over the element values, but sorting takes \(O(N\log N)\).

Execution times for different histogram algorithms are compared using the Merrimac cycle accurate simulator. Figure 4.16 shows that the execution time of privatization algorithms increases in proportion to both the range of the element values and the length of the stream, while for the algorithm using the hardware stream add&store it is only proportional to the length of the stream and not to the range of the element values. Getting the histogram of a stream using the hardware supported stream add&store operation takes less time than sorting the stream over the element values using merge sort (figure 4.17), which is a part of sort&count algorithm.

In addition to computing histograms, the scatter-add (indexed add & store) also works well in scientific applications where data is being pushed out to a shared data structure. For example in a particle-in-cell application, there is a phase in which the value of each particle is added to the value of the grid closest to that particle (right arrow of Figure 4.18). If there are more grid points than particles, scattering the values of particles into the grid points is more attractive than gathering nearby particles per grid and updating their value. Because several particles might try to update the value in a particular grid simultaneously, it is necessary on a conventional parallel architecture to lock the grids on each update in order to prevent a race. The stream add & store operation solves this problem without requiring a lock.

We have introduced a stream cache in order to improve performance on memory access patterns which are not regular but have temporal locality. For example when an element gathers the information of neighbors connected using irregular mesh structure, element data can be accessed
4.2. INTERCONNECTION NETWORK

Figure 4.16: Execution cycles of privatization and hardware scatter&add on histogram

several times within a short amount time.

Table 4.1.5 shows the performance improvement gained by using the stream cache with a sparse matrix vector multiplication application. One of the common ways to multiply sparse matrices with vectors is the compressed sparse row method (CSR), which compresses non-zero elements of rows by indexing their position. In our example application, the number of rows and columns are both 9978 and the number of non-zero elements in the sparse matrix is 441,676, so that the average number of non-zero elements per row or column is 44.26. In CSR, an element of the output vector is computed by the inner product of a row of the matrix with the input vector. Rows of the matrix are stored in the compressed form and the subsets of the vector for different rows are gathered by fetching the vector data using index streams. The performance of this problem is completely limited by memory bandwidth because three words of data are needed per every two arithmetic operations, one multiplication and one addition.

<table>
<thead>
<tr>
<th></th>
<th># of mem accesses</th>
<th># of cache accesses</th>
<th>cache hit rate</th>
<th>execution cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>without cache</td>
<td>1835856</td>
<td>0</td>
<td>N/A</td>
<td>666125</td>
</tr>
<tr>
<td>with cache</td>
<td>1835856</td>
<td>648112</td>
<td>93%</td>
<td>348345</td>
</tr>
</tbody>
</table>

Table 4.5: Compressed sparse row method results with and without cache

Because only the vector part has temporal locality and we can set the cacheablity of each stream, only one third of total memory accesses are set to be cached. Over 90% of cache accesses were serviced by the cache, and did not require off-chip data. This resulted in an execution time decreased of over 40%. The reason why the execution time decreases more than the percentage of cacheable memory accesses is because the memory access pattern of fetching the subset of the vector is not sequential and the performance degradation on non-sequential accesses is more severe than for sequential accesses when going from the on-chip stream cache to the off-chip DRAM.
4.2 Interconnection Network

The current sketch of the Merrimac interconnection network, shown in Figure 4.2, was described in detail in last year’s annual report. The network is four-way channel sliced. Each node connects to the network via four 5GByte/s full-duplex channels. The four channels connect to four completely separate network planes. This provides a degree of fault tolerance. If one plane fails, the system can continue operating (with reduced bandwidth) using the remaining three planes.

The basic building block of the network is a 48-port \( \times 5\text{GByte/s} \) per port router chip. The router supports a folded Clos [8] network\(^7\), providing 32 downstream ports and 16 upstream ports. Four copies of the router are used on each node board. Two downstream channels are connected to each of the 16 nodes on a board to provide flat bandwidth within the board. Eight of the upstream channels from each of these routers provide off-board links with \(1/4\) the aggregate node bandwidth. (The remaining eight upstream channels are unused). At the cabinet level, 32 routers (Packaged on eight boards) connect their 32 downstream links to the 32 boards in the cabinet. Each router

\(^7\) Sometimes called a Fat Tree [18].
connects to a particular upstream link of each of the 32 boards. All 16 of the upstream links of these routers are connected off-cabinet via optical links.\(^8\) At the system level, 512 of these routers\(^9\) are packaged 4 to a board in the system interconnect cabinet. These routers use only their downstream ports, each connecting to a particular upstream port of the 32 cabinets.

Over the past year, our research on interconnection networks has focused on three areas: topology, routing algorithms, and router micro-architecture. In the area of topology we have been investigating alternatives to the folded Clos topology of our straw-man network sketch. We have found that the optimal topology is very sensitive to assumptions about available signaling and packaging technology. For example, as the critical length of electrical signals becomes shorter, using Torus local networks becomes attractive. As another example, the optimal degree of channel slicing is dependent on chip and board pin limitations. To deal with this sensitivity to unknown packaging parameters, we are designing a methodology for topology selection — with supporting tools — rather than designing a specific topology. When complete, our methodology will accept a set of signaling and packaging technology parameters and output the optimal topology for those parameters. This will allow us to quickly select an optimal topology as packaging parameters shift.

For topologies like a Torus where multiple minimal and non-minimal paths exist to the des-

---

\(^8\)To save cost, the O/E and E/O modules are not installed in single-cabinet systems.  
\(^9\)Smaller configurations use fewer routers.
CHAPTER 4. MERRIMAC: THE STREAMING SUPERCOMPUTER

tination, occasionally (at high loads) some amount of traffic must be routed non-minimally to load-balance the network. We have developed a new method of adaptive routing that we refer to as **global adaptive load balance (GAL)**. Unlike previous adaptive routing algorithms that make routing decisions based on local information (typically channel queue depth), GAL uses global information in making routing decisions. This global information enables GAL to achieve the performance (latency and throughput) of minimal adaptive routing on benign traffic patterns while performing as well as the best obliviously load-balanced routing algorithms (Valiant or GOAL) on adversarial traffic.

Finally, we have started investigating the micro-architecture of **high-radix routers**. Most contemporary routers have a radix (or degree) of eight or less. That is, at each router, there are at most eight exit ports. Our straw-man network uses a radix-48 router. This high-radix enables us to build a low-diameter, and hence low-latency and low-cost, network. However, the high degree of the router raises a number of interesting questions of router organization. Current methods of constructing router switch fabrics and allocators do not efficiently scale to 48 ports. We are currently studying new methods to construct routers of this scale.

4.3 Software System

The software system of Merrimac is designed for ease of programming while achieving high performance on the Merrimac hardware. It is composed of three main parts: a programming language for expressing the problem, a compiler, and a stream virtual machine.

One of the main goals of the Merrimac project is to provide an easy-to-use programming system. There are two aspects of this goal that we plan to address. The first, which we have been working on over the last two years, is to define **Brook** the general streaming language for Merrimac. Brook allows the user to express the important characteristics of the program, which are parallelism and locality, in a way which is consistent, natural, and analyzable by the compiler. It is also important to note that Brook does not require a complete rewrite of the code base, as it is inherently compatible with C and provides a path to incrementally port an application to Merrimac. Essentially, only a minimal set of compute-intensive kernels need to be recoded, while most of the code base remains in its original C representation. We also provide a similar mechanism for Fortran programs through **Brooktran**, which is the Fortran90 equivalent of Brook. In the coming year we plan to start addressing the problem of automatic compilation of non-stream code to our streaming hardware.

The compiler takes code programmed by a user and performs high-level optimizations based on the SVM model and semantics, but targeted to a specific and detailed configuration of the system. After the high-level optimizations are performed, the code-generator produces the machine-code representation to be run on the hardware. This partitioning of the compiler into a high- and low-level sections, allows us to decouple some of the development effort and be able to leverage off existing compilation infrastructures.

The stream virtual machine presents the compiler an abstract, yet detailed, view of the hardware. This allows for general and effective high level optimization techniques to be common to multiple targets and system configurations, and represents a clean and uniform target that may be used by external compilers as well. In addition, the SVM provides a path for rapid software prototyping and user directed optimizations.

4.3.1 Brook: A Streaming Programming Language

An essential contribution of the streaming supercomputer project is defining an appropriate programming model for streaming computation. The programming platform must be able to capture all of the underlying hardware capabilities and performance, as well as provide a general interface
to map to a variety of existing architectures other than the streaming supercomputer.

During the first year of this project, we built a prototype of a new streaming programming environment entitled Brook. The Brook prototype defined many of the operations that we identified as fundamental for a streaming programming language. The second year of the project has been spent working with the compiler group to formalize the language into a mature, well defined, platform.

In the rest of this section, we will describe the Brook language and specifically highlight the aspects which evolved over the past year.

**Brook Semantics**

Brook is an extension of standard ANSI C and is designed to incorporate the ideas of data parallel computing and arithmetic intensity into a familiar, efficient language. The general computational model, referred to as *streaming*, provides two main benefits over traditional conventional languages:

1. **Data Parallelism**: Allows the programmer to specify how to perform the same operations in parallel on different data.

2. **Arithmetic Intensity**: Encourages programmers to specify operations on data which minimize global communication and maximize localized computation.

The basic computational unit in Brook is the *stream*. A stream represents an ordered list of elements requiring similar computation. The programmer executes functions, or *kernels*, on each element of the stream. Kernels operate on each element of an input stream and place the result of a computation on an output stream. The core of any Brook program consists of building streams of data and operating on that data with a variety of user-defined kernel functions. Kernels operate *implicitly* over the entire set of input streams. The kernel will execute once for each element in the input stream(s).

Kernel functions are declared similar to standard C-functions. An example kernel is shown below:

```c
void kernel foo (float a<>,
    out float b<>,
    float p) {
    b = a + p;
}
```

The parameter 'a' is the input stream, 'b' the output stream, 'p' a constant parameter. The body of the kernel foo will be executed for each input element of 'a', producing a new stream, 'b', which contains the set floats which are p larger than the corresponding 'a' elements. The <> operators specify the variable as a stream. Calling a kernel on a collection of elements is identical to calling any other C function, though the kernel is implicitly called on each element of the input stream(s).

```c
float x[] = {...}; // An array of floats
float a<300>;      // A stream of 300 floats
float b<300>;
float p = 3.02;
streamReadAll(a, x); // Set the stream a to the array x
```
foo(a, b, p);

Built on top of the concept of streams and kernels, Brook also permits the ability to perform *reductions*, computing a single value from a stream of elements, as well as providing *stream operators* which facilitate in the loading/storing and reordering streams and their elements for a particular computation.

**Brook Development**

This past year has been spent working with the compiler group and the application team to refine Brook from its prototype status to a complete, fully specified, language. This work culminated in the release of the v0.2 Brook spec. Below is an account of some of the developments that occurred this summer.

- The Brook C interface
  Previously, Brook was a stand-alone language requiring the application writer to port their entire application to use the Brook compiler. This posed a couple of challenges. First, many legacy applications were simply too large to even consider porting to Brook. By requiring the compiler to work with complete applications, we were forcing the compiler to support arbitrary C code as well as our streaming extensions. The solution was to separate the Brook portions from the existing C code with a distinct calling convention. This allows the application writer to only move the portions of the code which can benefit from streaming without having to run the entire application through our research compiler. Furthermore, the compiler can place some simple restrictions on the C language for the Brook routines. This allows the system to better analyze the streams and kernels which permits efficient scheduling of the hardware.

- Stream Shape
  Brook supports multidimensional streams in which a stream can consist of not just a 1D array of data but an nD collection. However previously, the *shape*, or dimensionality, of a stream was a mutable property of the variable. One could change the shape of any stream variable with the reshape command. This was both difficult for application developers to understand given the lack of any C parallels, and challenging for the compiler group to work with. The solution was in incorporate the shape into the stream type.

  ```
  float s<300, 400, 200>;
  ```

  A stream of 24 million floats in the shape of 300x400x200.

  Linking the stream shape to the type provided clarification to C veterans as well as simplified the task for the compiler group. A survey of the application writers discovered that there was little use for the reshape command. However, the current version of the language in which stream shapes are required to be compile-time constants is overly restrictive. We expect to allow stream shapes to be specified statically by run-time variables in the future.

- Stream declaration syntax
  The new stream declaration syntax also clarified the stream declaration syntax which was awkward with the previous stream keyword syntax.
4.3. SOFTWARE SYSTEM

float a<>; // Stream of floats
float b<>[3][2]; // Stream of float array[3][2]
float c[2]<>; // Array of streams ***
float d<><>; // Stream of streams ***
typedef float mytype[3][2];
mytype e<>; // Stream of float array[3][2]
typedef float mystreamtype <>[3,2];
mystreamtype f; // Stream of float array[3][2];
mystreamtype g<>; // Stream of streams ***
mystreamtype h[2]; // Array of streams ***

***: Not currently supported in Brook but may be included in future versions.

- Reductions

Before this year Brook supported reduction arguments to kernels which permitted a read/modify/write variable for computing arbitrary values from a stream. It was discovered during the compiler implementation that the existing mechanism seriously limited the parallelism achievable with the hardware. As a result, the reduction mechanism was redesigned.

Reductions can be performed from within kernel functions via reduction variables or via user specified reduction functions. Below is a simple reduction which computes the sum of a float stream using a kernel reduction variable.

```c
void kernel sum (float a<>,
    reduce float result) {
    result += a;
}
```

The reduction variable, `result`, is specified with the reduce keyword. A kernel function is permitted to have multiple reduction variables declared in the argument string. Operations on reduction variables are limited to permit the compiler the opportunity to parallelize the computation. This example updates the value of the reduction variable with the incoming stream of floats to produce the sum of all the elements in the input stream `a`. Though the reduction can be perceived as sequential, Brook does not guarantee the order of operations on reductions. Though addition is both commutative and associative theoretically, due to the IEEE fp32 limitations, the reduction result is an approximation of the mathematical result.

Reduction functions are user specified reduction operators which can be called from within a kernel function or directly from the Brook code to perform a reduction directly on a stream. Reduction functions only support reductions which are both associative and commutative. The compiler must be able to compute the reduction in any order. Reduction functions are specified similar to kernel functions with few additional restrictions. Below is a sample reduction function which computes the sum of a float stream.

```c
void reduce sum (float a<>,
    reduce float result) {
    result = result + a;
}
```
CHAPTER 4. MERRIMAC: THE STREAMING SUPERCOMPUTER

Specification & Compiler Development

Brook has definitely matured in the past year to a point in which it is a robust data parallel language. The first year of the project, Brook was largely driven by application developers. The year, the language has been driven most by the compiler team which has helped center the language which is complete and analyzable. Of course, there will certainly be more aspects of the language that we may find deficient as the compiler gets closer to functioning output.

4.3.2 Brooktran

Brook is an extension of C which allows incremental porting of pre-existing C codes: computationally intensive functions can be individually replaced with kernels to take advantage of the stream processors (a "kernel" is a piece of code that will be executed on the streaming processor unit). This incremental “streamification,” similar to the incremental parallelization possible with OpenMP, greatly simplifies porting an existing C code to Merrimac. The code will run with no changes on Merrimac, and increasing performance can be obtained by converting regular functions to kernels.

For pre-existing codes written in Fortran the situation is completely different. The whole code needs to be rewritten in C and Brook, with their many differences from Fortran, making the task even harder than, say, parallelizing a serial code using MPI. Fortran is still very popular in the High Performance Computing community. The national laboratories, national agencies (such as NASA), universities, and many industries have millions of lines of existing, valuable Fortran codes. This investment in Fortran can be expected to continue with the advent of the Fortran 90 and 95 standards, which include modern memory management and data and control structures. Porting all these codes to C/Brook is not feasible, particularly considering the validation efforts that have been spent bringing the codes to their current state of acceptability. Some of these codes are used for mission critical tasks which require their results to have a very high level of reliability.

The purpose of Brooktran, a streaming extension of Fortran, is to give an incremental porting path to the Fortran community. In order to have an interface consistent with the C counterpart, we will adapt certain Fortran 90 features, such as the "intent" attribute, to the language. This also facilitates use in a mixed-language environment.

The kernels are written using a Fortran syntax and have the same constraints as Brook kernels. As in Brook, the setup of streams is done through library calls. To add streaming capabilities to Fortran, we need a few new keywords and to define some rules for kernel invocation.

The following keywords have been added to Fortran:

- **stream**: used to define a stream. It is a native compound object like array. By default streams are one dimensional. Other dimensionalities are defined by the dimension given in the declaration, e.g.,

  \[
  \text{stream, real, dimension,:,:} :: a
  \]

  defines a two dimensional stream of reals. For streams that are generated from stencil or group operators, we can specify the shape of each element, e.g.,

  \[
  \text{stream, real, dimension,:,:} :: b(3,3)
  \]

  defines a stream of $3 \times 3$ reals which could be used to access multiple elements of a stream via a streamStencil call.
• *kernel*: used to specify that a function or subroutine can be executed on the streaming processor unit. There are certain restrictions associated with kernels; for instance, all the arguments of a kernel must have their access specified explicitly, and no common blocks or modules may be used inside a kernel.

Stream manipulation functions and operators

As in Brook, stream manipulations are performed through a library of stream operators. For example if the source is an array X(100,200) the following code will generate a stream Y from X with 100*200 elements:

```
call streamRead(Y,X,100*200)
```

We can also use the intrinsic array processing of Fortran 90 to select a specific subset of an array:

```
Y2 = streamRead(X(51:100,101:200),50*100)
```

The superior array syntax of Fortran 90 should permit expressing most of Brook's streamDomain operations using array syntax.

Kernel functions or subroutines:

Kernel functions or subroutines are declared like regular Fortran functions/subroutines by placing the “kernel” keyword first. The restrictions on the arguments of the call are the same ones present in Brook. The Fortran “intent” attribute is used to specify the access attributes of streams, e.g., read only or write only. Each argument of a kernel must have its access specified explicitly as being either “in”, “out”, or “reduce”, where “reduce” specifies a reduction variable to accumulate sums, products, etc.

```
program sample
real, allocatable, dimension(:):: a,b,c
stream, real, dimension(:) :: stream_a, stream_b, stream_c
integer :: n
real :: total_sum
n = 1000
allocate(a(n),b(n),c(n))
....... 
call streamRead(stream_a, a, n)
call add_array(stream_a, stream_b, stream_c, total_sum)
call StreamWrite(stream_c,c,n)
end program sample

kernel subroutine add_array( a ,b , c , sum)
    stream, real, intent (in) :: a,b
    stream, real, intent (out) :: c
    real, intent(reduce):: sum
    c = a + b
    sum = sum + c
end subroutine add_array
```
The initial development of Brooktran was done using the Open64 compiler infrastructure to
generate an additional phase able to parse Brooktran and produce WHIRL intermediate code.
With the decision to switch to a different compiler infrastructure we need to revise our plan.
The effort is now focused on producing a Brooktran to Brook translator. One possible means of
producing such a translator is to continue to use the Open64-based Brooktran parser to produce
WHIRL, and to enhance the WHIRL-to-C pass of Open64 to produce Brook.

4.3.3 Compiler

The Merrimac compilation system has two main purposes. The first is to compile a Brook or
Brooktran program directly to a Merrimac executable. The second is to provide the programmer
with a system for debugging and verifying the code, as well as performing fast analysis of program
performance and behavior without use of the Merrimac hardware. This second goal is accomplished
by compiling down to standard C combined with a run-time library as well as the SVM and using
a fast SVM simulator native to the development hardware system.

In the first year of the project we took the approach of quick experimentation in order to
validate the general ideas incorporated into the system – the stream programming language Brook,
and the streaming hardware specialized for scientific applications. In the last year we took a step
back, analyzed our results, started a design plan for the permanent compiler, evaluated compiler
framework options and started the implementation.

In the rest of this section we will describe the evolution of the compilation system, summarize
the current implementation status, and detail the phases and analyses planned.

Compiler Evolution

The initial compilation system was put together in order to quickly evaluate and validate the
general ideas of supercomputing with streams. For this purpose it was based on a collection of
tools that were relatively easy for us to modify to suit the task of performing simple non-optimizing
compilation of Brook. The system comprised of a meta-compiler [7] and the Imagine programming
system [21]. The meta-compiler allowed us to parse the original version of Brook and perform very
simple code transformations based solely on information gathered from parsing the code line by
line. This allowed us to take a Brook program and produce somewhat optimized StreamC and
KernelC code, which was then run on the Merrimac hardware simulator using modified versions of
the Imagine StreamC compiler and kernel scheduler as described in the 2002 annual report.

We used this initial system, and with some manual assistance, compiled the two original driving
applications – streamFLO and StreamMD. We then used these results as part of the proof of concept
of the Merrimac system and scientific stream computation in general. After achieving this initial
goal we evaluated the suitability of our initial compilation system and decided we had to abandon
it in favor of a more solid compilation infrastructure. The two main reasons for this were due to
limitations of the meta-compiler:

- The meta-compiler only accepts syntax that is a small deviation from C, limiting the develop-
  opment of Brook and Brooktran
- The tools available within the meta-compiler framework were not enough to perform the
global analysis required for producing efficient Merrimac code

The experience we gained was invaluable, both in providing feedback to the application writers
and Brook developers and in understanding the types of analyses and optimizations required for
compiling to the Merrimac architecture. The most important lesson we learned is that we need to
base our compilation system on a solid infrastructure that provides tools for global analysis.
4.3. SOFTWARE SYSTEM

The next step we took is to define a set of criteria by which we could evaluate a number of available compilation frameworks and choose the one most suitable for the Merrimac compiler. Our evaluation criteria included availability of a Fortran frontend and ability to modify the C and Fortran syntax, extensibility of the intermediate representation to include streaming information, amount of infrastructure already in place to perform the required analyses, and general support in the industry and academic community. After examining several compilation frameworks (GCC [1], TenDRA [6], SUIF [4][5], and Open64/ORC [3][2]) we came to the conclusion that Open64 was the most suitable framework choice for a long-term compilation infrastructure.

In collaboration with Alan Wray of NASA Ames Research Center, we proceeded to modify Open64 to serve as the Merrimac compiler. We developed a complete Brook and Brooktran frontend, defined a streaming intermediate representation, and began work on a few of the necessary tools and analyses phases. Before completing an end-to-end simple compilation we halted work on the Open64 platform.

There were two reasons for shifting our focus again. The first was that the Brook specifications were changing rapidly and we had to wait for both the semantics and syntax to stabilize. The second reason was that Reservoir Labs was contracted by DARPA to develop a Brook to SVM compiler. We decided to leverage their experienced compiler team and re-target their compiler to Merrimac.

Merrimac Compilation System Design

As mentioned before, the Merrimac compilation system takes a Brook or Brooktran program as input and supports output for three different targets:

- **Brook Run Time (BRT)** – the Brook runtime allows for rapid development of Brook and Brooktran programs. The source code is compiled to a C version of the program with calls to a library which implements the functionality of Brook. The C code is then compiled to an executable using a native C compiler (Figure 4.20).

- **Stream Virtual Machine (SVM)** – the SVM sets an intermediate target with a representation that is common to all stream architectures. It can be used as a debugging tool for the application and compiler algorithms, for fast simulations of large multi-node systems, and most importantly allows for a single set of high-level compilation phases to be shared by several physical machine targets.

The compilation path involves several necessary phases and a set of optional optimizations. These phases are designed specifically with the Merrimac hardware in mind, but the compiler output a C like program that conforms to the SVM semantics and syntax as described in Section 4.3.4 (Figure 4.21).

- **Merrimac** – compiling to the Merrimac hardware involves producing two separate types of code: scalar and stream instructions that call kernels, and kernel code that is run on the compute clusters. The scalar/stream code path is similar to the one used to produce SVM code. To produce the kernel micro-code communication scheduling is performed. Initially, we will use an external tool to perform this. The Merrimac compiler will output kernel code as KernelC [21] that will feed into the kernel scheduler. At a later stage the communication scheduling ([22]) will be integrated into the compiler, which will then directly output micro-code. The main reason for integrating the kernel scheduler is to provide tighter collaboration and more effective feedback between the high-level and low-level optimizations and transformations. Both paths are depicted in Figure 4.22.
The common phase to compiling code for all three targets is the *frontend parsing*. In this phase the program is transformed down to a representation that is tuned for analyses and optimizations. This very high level intermediate representation is referred to as an *abstract syntax tree* (AST) and is completely equivalent to the original source. The frontend may also perform simple optimizations that do not change the semantics of the program.

The next set of compilation phases *lowers* the AST representation so that it conforms to the semantics and architectural limitations of the target. We identified a number of necessary phases as well as optimizations that can be applied. A brief description of these is provided below. Note that compiling down to C with a run-time library requires none of these phases.

**Conditionals**

Conditional control flow, other than loops, within kernels is not allowed on the Merrimac hardware. Therefore, the compiler must transform conditional code into either predication (software selects) or conditional streams [11]. The simple non-optimizing phase will simply expand all conditionals into software predication using Merrimac’s *select* instructions.
4.3. SOFTWARE SYSTEM

Brook Operators

A major compilation phase is to translate the Brook operators down to machine primitives. There are three main types of operators, which are discussed below.

- **Multi-dimensional Operators**
  Multi-dimensional streams and operators are commonly used in Brook to efficiently express algorithms dealing with physical data. The hardware only supports single-dimensional streams, and efficiently converting between the two is critical to achieving high performance. In the last year we developed compiler algorithms for handling multi-dimensional streams using hardware with a conventional stream register file (SRF). After analyzing the performance of these automatic transformations we realized that a more advanced method that also takes advantage of Merrimac’s indexable SRF is required for achieving the desired performance. We are currently exploring such algorithms as well as studying various SRF and cache organizations in the hardware (Section 4.1.2).

- **Memory Operators**
  All Brook memory operators have are directly and fully supported by Merrimac’s address generators and memory system.

Stream Lengths

Stream Scheduling is highly dependent on the stream lengths. This phase attempts to derive the lengths of streams based on the flow of the code, categorizes the streams, and assigns them properties as follows:

- **Constant length** – the stream length is constant. The constant may be both a compile-time or a load-time constant. The initial version of the compiler will only be able to deal with streams of this type.
- **Relative length** – the stream length is unknown, but it is a constant factor of the length of another stream.

- **Variable length** – the stream length is completely dynamic. If the compiler can derive a maximum or expected length that information is stored as well.

Note that the stream lengths and properties are per semantic stream in the program, and not per named stream. If a stream’s property or length changes during the program it is assigned different properties at different points in the intermediate representation.

**Double Buffering**

The execution model of Merrimac only allows a kernel to run when all its input streams are loaded into the SRF and there is enough space allocated in the SRF for the outputs. When the inputs are too long, or the outputs are not bounded, the system must orchestrate memory operations with the multiple invocations of the kernel. Double buffering is the simplest general way to deal with both long streams, and variable length streams. The amount of work this phase requires depends on the underlying machine capabilities. Merrimac supports transparent double-buffering from the kernel perspective and requires the compiler to insert the appropriate memory operations and kernel (re)invocations instructions.

**Scalar/Stream Synchronization**

In Merrimac, there is one instruction stream controlling both the execution of scalar code and stream code (stream memory operations and kernel invocations). However, the completion of the two types of instructions is asynchronous, where memory operations (scalar and stream) as well as kernels have a dynamic completion time. Care must be taken to synchronize between the issued instruction stream and the completion of dynamic latency instructions. For example, a scalar read to a memory location that is being scattered to cannot proceed until the scatter operation completes. This compilation phase analyzes the Brook code (flow analysis) to determine instruction dependencies and possibly insert synchronization instructions (barriers).

**Memory Allocation**

The compiler must handle the memory allocation for both blocks and streams. Block allocation will be handled transparently using standard C mechanisms, but calls to allocate memory for backing up streams or for temporary memory buffers must be inserted into the program representation.

**Multi-node Support**

Multi-node support requires three basic operations, which are straightforward to implement in a non-optimized manner:

- **Data partitioning** across the nodes of the system. As a first implementation this partitioning can be done arbitrarily.

- **Work partitioning** across the nodes. Again a simple implementation would be to assign work by the location of one of the output or input elements.

- **Synchronization** between the concurrently running nodes. A conservative and simple implementation is fairly straightforward and at worst will synchronize after every memory store.

**Stream Scheduling**

Stream scheduling encompasses SRF allocation, stream memory operation scheduling, and
kernel invocation scheduling. It is a fairly complicated process that has been studied intensively under the Imagine project. The plan for the Merrimac compiler is to adopt much of that body of work [13].

Conditionals Optimization
The non-optimized conditional phase transforms all conditionals into software predication. The Merrimac hardware also provides support for more sophisticated ways of dealing with conditionals – conditional streams and conditional load-balancing. A few suggestions for compiler algorithms that automatically optimize for these techniques, as well as predication, will be presented in [12].

Strip-Mining Optimization
While double-buffering is a generic technique for dealing with long and variable length streams, it is not necessarily the most efficient. The problem with double-buffering is that all the input and output elements are communicated via memory, thus limiting performance to the memory bandwidth. The compilation system can take advantage of producer-consumer locality between kernels to reduce this memory traffic. The method of doing so is referred to as strip-mining or blocking [20].

Software Pipelining Optimization
Automatic software pipelining and unrolling of stream-level loops can be a very effective technique of increasing cluster utilization by overlapping memory transfers with kernel invocations.

Kernel Partitioning Optimization
This pass analyzes a kernel and, if necessary, partitions it into two or more simpler kernels. The partitioning decision is based on balancing the amount of work that needs to be redone, the amount of data stored and loaded, and ensuring that the kernels meet the resource limits of the hardware. This pass requires flow analysis as well as good models of the clusters.

Kernel Coalescing Optimization
This phase is similar to kernel partitioning, and aims to increase short-term producer-consumer locality and decrease kernel invocation overheads.

Operator Combining Optimization
Combine and collapse a sequence of Brook operators and kernels. Includes optimizing reductions and automatic use of Merrimac’s ScatterOp capabilities.

Record Partitioning Optimization
Record partitioning (and coalescing) can be used to reduce memory traffic when a series of kernels is called that only uses a subset of the record’s fields. This optimization phase allows the programmer to specify records in a way that makes sense from a software engineering perspective, and relies on the compilation system to optimize memory traffic and SRF usage.

Multi Node Optimization
Better algorithms for automatic partitioning and removal of synchronization points. We have not addressed these issues at all in our compiler development thus far.

Stream Scheduling Optimization
This phase is an enhancement of the simple stream scheduling and includes reordering stream operations, kernel invocations, and synchronization points to minimize stalls. This phase will also insert hints for using the stream cache and deal with the coherency issues that arise from replicating data on different nodes.
The final compilation step is code generation (back-end), which lowers the compiler intermediate representation down to machine instructions. In the case of the Merrimac compilation system, three different back-ends are provided as explained in the beginning of the section. The most interesting of these code-generators is the one producing Merrimac cluster μ-code. While it is desirable for this code to be generated at the same time as the Merrimac stream and scalar codes, the initial infrastructure will use a stand alone kernel scheduler to perform this step.

Current Status

The current compiler based on the Reservoir infrastructure is at a very early stage. The system can parse the latest specification of Brook into an intermediate representation, and produce SVM code for a limited set of specific examples. We are currently in the process of spiking the compiler (running a single program through the compiler from parser to code generation) using the streamFEM application, which has been recoded to conform to the latest version of Brook.

The compilation infrastructure does not currently support Brooktran and our plan is to use our Open64 frontend to parse Brooktran code, and to modify the Open64 C-printer so that it can output Brook. In this way we can form a path from Brooktran to SVM and to the hardware with little effort.

Another piece on which we are actively working is the code generation to Merrimac. Our initial target is to use an external tool to convert the compiler’s SVM output into a C-like representation of the Merrimac instruction set (this type of representation has been successfully tested using Imagine).

4.3.4 Stream Virtual Machine

The Stream Virtual Machine (SVM) is a machine-independent model of a parallel streaming computer that is used as an intermediate representation in the compilation process from a high level stream language like Brook down to a specific stream architecture. By compiling first to the SVM and then from the SVM to a target architecture as illustrated in Figure 4.23 we can make our software portable across many different target architectures. Only the back-end of the compiler needs to change to compile code for a new target architecture. All of the applications, and the front-end of the compiler are completely portable.

The SVM has been adopted by the DARPA PCA project (Polymorphous Computing Architectures) as the stream computation model. Stanford Smart Memories, MIT Raw, UT Trips and USC’s Monarch are the 4 processor architecture from that program that will implement the SVM. As part of this contract, Reservoir Labs, a company is implementing a compiler that takes Brook and generates SVM. As described in Section 4.3.3, we are collaborating with Reservoir Labs to adapt this compiler to target Merrimac as well.

SVM Syntax

The SVM syntax is also based on C but is much lower level than Brook. At the SVM level, the problem is partitioned and resources, such as the Stream Register File, are allocated. The SVM allows for stream computation to be both in the time and in the space domain, i.e. you can have kernels run simultaneously on different stream processor and also to time-multiplex kernels on a single stream processor. For Merrimac, the current implementation plan is to run an identical kernel on each stream processor, each operating on a different part of the data and time multiplex these kernels.

The SVM syntax consists of special C structures and function calls that allow a regular processor to control a slave stream co-processor. The regular processor runs control code while the stream
co-processor runs kernels with their own special API to interface to streams.

The SVM syntax has gone through several review cycles as part of the DARPA PCA program. Now all the teams that are using the SVM have agreed on the syntax and it’s a stable API.

**SVM Machine Model**

In order to describe a particular machine, like Merrimac, Smart Memories, or another, in terms of a Stream Virtual Machine, we had to agree on a way to represent the essential characteristics of a SVM. This information will be used by the compiler to produce optimized code for this particular SVM, and by the SVM simulator to get application performance estimates.

The SVM machine model defines both the XML file format and how to describe a particular architecture. Processors have to be defined as regular, stream or DMA processors with many other details on their performance. Memories are described as caches, FIFOs, or random access memories with their capacity and bandwidth. To connect processors and memories, network links are characterized by their connected components, bandwidth and latencies.

**SVM Simulator**

Since SVM code is just vanilla C with special function calls for the stream portions, it is possible to compile SVM code with the right library which implements the SVM calls. This is basically what the SVM simulator is, an implementation of the SVM special functions that allows to run SVM code on a normal workstation to test it for functionality and to estimate performance.

In the last year, the SVM simulator has gone through the changes in syntax in the SVM API, and has been extended from a single node simulator to a multi-node simulator. It is a threaded simulator that uses the pthread thread library available on most UNIX and Linux platforms.

The SVM simulator first sets up the appropriate threads to simulate the machine by parsing the machine description file. From the machine description it also knows some basic information about the performance of the machine like bandwidth numbers from network links and memories. This information will be used to estimate performance of the application on the target machine.

The SVM simulator is intended to be used to simulate large multi-node configurations of Merrimac. The kernel run times based on the kernel scheduler will be used to tune the estimates of the kernel run times. Memory bandwidth is also monitored and the bottlenecks between these two can be identified and taken into account in the total runtime estimate.

---

![Compilation Process Diagram](Figure 4.23: Compilation Process)
4.4 Merrimac Applications

One of the strengths of the Merrimac project is the close interaction between application developers and the hardware and software group. This interaction has helped to ensure that the Brook streaming language provides the necessary language constructs used frequently in scientific applications. We are currently working on 4 applications: StreamFEM, StreamSPAS, StreamFLO and StreamMD. As the language and the tools become more robust, we intend to add more applications to the list in order to cover a broader range of scientific disciplines.

StreamFEM-3D and StreamSPAS are two applications utilizing unstructured meshes that have been developed specifically for evaluation of the Brook streaming language and the Merrimac hardware design. StreamFEM-3D is a three-dimensional discontinuous Galerkin (DG) finite element application for first-order systems of conservation laws on tetrahedral meshes. StreamSPAS is an application for the design and evaluation of streaming language implementations of two sparse matrix algebra pervasive in scientific computing: sparse matrix-vector products and sparse complete/incomplete matrix factorization.

StreamFLO is a two-dimensional finite volume solver for the Euler equations that uses a non-linear multigrid. StreamFLO utilizes structured meshes. StreamMD is a code for molecular dynamics simulations, based on GROMACS [28] a popular open-source package.

4.4.1 StreamFEM-3D

StreamFEM-3D extends the original two-dimensional StreamFEM application to three space dimensions. StreamFEM-3D implements the discontinuous Galerkin finite element method for solving a system of $m$ coupled first-order differential equations in three space coordinates and time. Specifically implemented are DG formulations for the Euler equations of gasdynamics and the magnetohydrodynamic (MHD) equations. Let $u(x, t) : \mathbb{R}^3 \times \mathbb{R}^+ \rightarrow \mathbb{R}^m$ denote the dependent solution variables and $f(u) : \mathbb{R}^m \rightarrow \mathbb{R}^{m \times 3}$ be the flux vector. The prototype Cauchy problem is then given by

\[
\begin{aligned}
\left\{ \begin{array}{l}
u_t + \sum_{i=1}^{3} f_{x_i} = 0 \\
u(x, 0) = u_0(x)
\end{array} \right.
\tag{4.2}
\end{aligned}
\]

The StreamFEM-3D implementation of the DG method utilizes piecewise polynomial approximations of degree $0 \leq k \leq 3$ in space and of degree 0 in time with no continuity between space-time elements such that for a single time slab $I^n = [t^n_+, t^{n+1}_+]$ with $[0, T] = \cup_n I^n$ and spatial element $K$

\[
\mathcal{V}^h = \left\{ u^h \mid u^h_{|K \times I^n} \in \left( P_k(K) \right)^m \times P_0(I^n) \right\}.
\]

The DG method is then succinctly stated in variational form for a single time slab.

Find $u^h \in \mathcal{V}^h$ such that for all $w^h \in \mathcal{V}^h$

\[
B(u^h, w^h)_{DG} = 0
\tag{4.3}
\]

and

\[
B(u, w)_{DG} = \int_{I^n} \int_{\Omega} (-u \cdot w_t - f^i(u) \cdot w_{x_i}) \, dx \, dt
\]
\[
+ \int_{I^n} \int_{\Omega} \left( w(t^{n+1}_+) \cdot u(t^{n+1}_+) - w(t^n_+) \cdot u(t^n_+) \right) \, dx
\]
\[
+ \int_{I^n} \sum_{e \in \mathcal{E}} \int_e \left( w(x_+) - w(x_-) \right) \cdot h(u(x_-), u(x_+); n) \, dx \, dt
\]
where $h(u(x_-, u(x_+))$ denotes a numerical flux function. Boundary conditions for solid walls and characteristic boundaries (not shown here) are included in the actual implementation. Implicit “mass” matrices are avoided by the use of $L_2$ orthogonal basis representations in each element so that the time stepping becomes explicit.

Figure 4.24: Single Merrimac chip performance for the StreamFEM-3D application. Charted cases are: 0-E (p=0, Euler), 0-M (p=0, MHD), 1-E (p=1, Euler), 1-M (p=1, MHD), and 2-E (p=2, Euler).

The computational kernels of StreamFEM-3D have been ported to StreamC/KernelC so that precise hardware performance can be measured using a suite of Merrimac simulation tools originally developed for the Stanford Imagine hardware. The preliminary results for a single Merrimac processor chip are charted in Fig. 4.24. A sustained performance is achieved in the computational kernels that is approximately 50% of the peak 64 GFLOP performance of a single Merrimac chip. Further refinements (both hardware and software) will improve this further.

The Brook implementation of the two-dimensional StreamFEM code has been given to the Reservoir Labs Inc. to test and otherwise facilitate their development of a compiler for the Brook language.

### 4.4.2 StreamSPAS

StreamSPAS (Streaming SParse Algorithm Suite) is a application package under development which explores streaming language implementations and Merrimac hardware performance for two ubiquitous sparse matrix algorithms in computational science: sparse matrix-vector products (current development) and sparse complete/incomplete matrix factorizations (future development). The current version of StreamSPAS contains a general test suite framework which generates sparse matrices based on three-dimensional finite element discretizations of elliptic PDEs using conventional $p$-order ($p=1,2,3$) Lagrange tetrahedral elements with full continuity. Four common compressed sparse structures have been implemented in StreamSPAS:

- Compressed sparse row (CSR)
- Compressed sparse column (CSC)
- Hypergraph edge storage (HES) for pattern symmetric matrices
- Element-by-element storage (EBES) for FEM matrices
The CSR and CSC storage schemes are conventional storage schemes for sparse matrices, see for example Saad [25]. The hyperedge storage scheme (HES) assumes pattern symmetry of the sparse matrix. HES stores pairs of matrix elements symmetrically located about the diagonal of the matrix. The amount of storage needed in the hyperedge storage scheme is identical to the CSR and CSC storage schemes. The element-by-element scheme (EBES) assumes a finite element method and represents the global matrix-vector product as the sum of smaller elementwise matrix-vector products. EBES has larger memory storage requirements than found in CSR, CSC and HES but algorithms utilizing the EBES scheme exploit the performance benefits of performing dense elementwise matrix-vector operations. Each data structure suggests a different sparse matrix-vector algorithm. Let $A$ denote an $m \times m$ matrix with nonzero entries stored using some sparse representation. The task at hand is the calculation of the sparse matrix-vector product

$$A\bar{x} = \bar{y}$$

for densely stored vectors $\bar{x}$ and $\bar{y}$ each containing $m$ entries.

- **Algorithm SpMatVecCSR.** Let $\bar{r}_i$ denote the $i$-th row of the matrix $A$,

$$A = \begin{bmatrix} \bar{r}_0 \\ \vdots \\ \bar{r}_{m-1} \end{bmatrix}.$$  

The matrix-vector product is computed as $m$ independent dot products,

$$y_i = \bar{r}_i \cdot \bar{x}, \quad i = 0, \ldots, m - 1.$$  

The streaming processor implementation of this algorithm requires memory gather operations.

- **Algorithm SpMatVecCSC.** Let $\bar{c}_i$ denote the $i$-th column of the matrix $A$,

$$A = [\bar{c}_0, \ldots, \bar{c}_{m-1}].$$  

The matrix-vector product is computed as a linear combination of matrix columns,

$$\bar{y} = \sum_{i=0}^{m-1} x_i \bar{c}_i.$$  

The streaming processor implementation of this algorithm requires scatter-op operations to memory.

- **Algorithm SpMatVecHES.** Let $e$ denote a hyperedge in the graph of the sparse matrix. The matrix-vector product is computed as

$$A\bar{x} = A_{\text{diag}}\bar{x} + \sum_{e \in \text{hyperedges}} A_e \bar{x}_e$$

where $A_e$ and $\bar{x}_e$ are restrictions of $A$ and $\bar{x}$ to an edge of the hypergraph. This algorithm utilizes a mixture of gather and scatter-ops to memory when implemented on the Merrimac hardware.

- **Algorithm SpMatVecEBES.** Let $k$ denote an element in a FEM mesh. The matrix-vector product is computed as

$$A\bar{x} = \sum_{k \in \text{mesh}} A_k \bar{x}_k$$
where $A_k$ and $x_k$ are the element contributions to the global matrix and vector. This algorithm requires more storage and has extra arithmetic operations when compared to the previous algorithms but has elementwise dense matrix-vector operations which perform well on the Merrimac hardware.

These algorithms have been ported to StreamC/KernelC so that hardware performance can be measured using the Merrimac hardware simulation tools. Figures 4.25-4.26 chart performance of these algorithms (note logarithmic scale axis) when compared to a Pentium 4 processor running at 2.8Ghz. Both the Merrimac and Pentium 4 calculations for algorithms such as $\text{SpMatVecCSR}$ are performance limited by the bandwidth to memory. In this case, the 20x improvement in memory bandwidth of the Merrimac architecture over the Pentium 4 architecture translates into a similar overall performance gain. Future work will include matrix element compression strategies for reducing the memory bandwidth requirements and increasing the overall performance on both the Pentium 4 and Merrimac architectures.

The use of unstructured meshes in StreamFEM and StreamSPAS necessitates irregular data access from/to main memory. The Merrimac hardware design includes full support for hardware gather/scatter operations that are heavily utilized by these applications. In addition, a stream cache and indexable SRF have been added to the hardware design. The StreamFEM and StreamSPAS applications provide two of the most severe tests of these added hardware features and the entire memory subsystem of the Merrimac design.
4.4.3 StreamFLO

StreamFLO is a finite volume 2D Euler solver that uses a non-linear multigrid algorithm. The original FORTRAN code (FLO82) was written by Prof. Jameson and this approach is used in many industrial and research codes. The choice of the code was motivated by the need for a complete application that was representative of a typical CFD application, without unnecessary complexity.

Governing equations and discretization

The equations solved are the 2D Euler equations written in conservative form:

$$\frac{d}{dt} \iiint_{\Omega} w \, dx \, dy + \oint_{\partial \Omega} (f \, dy - g \, dx) = 0,$$

where $w = \{ \rho, \rho u, \rho v, \rho E \}$ is the vector of conserved flow variables, and $f, g$ are the Euler flux vectors

$$f = \begin{cases} \rho u \\ \rho u^2 + p \\ \rho uv \\ \rho uH \end{cases}, \quad g = \begin{cases} \rho v \\ \rho uv \\ \rho v^2 + p \\ \rho vH \end{cases}.$$

This set of equations is completed by an equation of state for the pressure:

$$p = (\gamma - 1) \rho \left[ E - \frac{1}{2} (u^2 + v^2) \right].$$

A cell-centered finite-volume formulation is used to solve the equations together with multigrid acceleration. Time integration is performed using a five stage Runge-Kutta scheme.

$$\frac{d}{dt}(w_{ij} \, V_{ij}) + R(w_{ij}) = 0.$$

On the fine mesh, a symmetric limited positive H-CUSP scheme [26, 27] is used for artificial dissipation, while on the coarse meshes, a standard JST (Jameson-Schmidt-Turkel) scheme is employed.

Results

We have two implementations of the Brook language: one is a compiler that translates Brook to C and then links with a set of runtime libraries for the stream manipulation. The other is a compiler that produces code for a cycle-accurate simulator based on the Imagine tools. We use the first implementation to check correctness, perform language studies and to run on standard hardware (the compiler is based on gcc and runs on several platforms). The second tool is used to predict performance of the code on actual hardware (when it becomes available). Both tools are still under development and the optimization is far from optimal. The results from the first simulations are very encouraging. The schedule of the kernel for the computation of the diffusive flux using a symmetric limited positive H-CUSP scheme is shown in figure 4.27. This kernel currently runs at 37 GFlops, 58% of the peak performance. Improving the set up of the stencil operator (that is now using only 20% of the resources) will bring the sustained performances up to 50 GFlops, close to 80% of peak.

A run with a single multigrid level was completed on the hardware simulator, and the whole code was able to sustain 11.4 GFlops. This simulation was run on a version of the simulator that included four multiply/add units per cluster (for a peak performance of 64 GFlops/node) rather than the
4.4. MERRIMAC APPLICATIONS

Figure 4.27: Kernel schedule for the H-CUSP dissipative flux calculation: Cycles are marked vertically on the left and the different columns correspond to the various resources of the SIMD cluster used. The 4 left-most columns in the magnified section are the 4 multiply and add units, the fifth column represents the lookup table for the inverse and square-root operations. The 3 right-most columns are related to the intracluster communications.

four integrated Multiply and Add units (128 GFlops/node) that are in the current design. In this result, divides are counted as single floating point operations, even though each divide requires several multiplication and addition operations when executed on this hardware. The performance will double if we counted all the multiplies and adds required for divisions as well.
Future work

A 3D Navier-Stokes solver is being developed to further study the validity of the approach. The solver contains a large portion of the execution kernels of the three-dimensional, Reynolds Averaged Navier-Stokes flow solver TFLO and is intended to provide performance benchmarks for realistic, industrial-strength flow applications. This new solver will be written in a new variant of the Brook language called Brooktran. Brooktran is identical to FORTRAN 95 in its syntax and contains extensions to handle streaming constructs.

4.4.4 StreamMD

Molecular Dynamics is a technique to compute kinetic and thermodynamic properties of a molecular system. We used GROMACS[28] as the basis code for our simulation. GROMACS is a versatile package to perform molecular dynamics, i.e. simulate the Newtonian equations of motion for systems with hundreds to millions of particles. It is primarily designed for biochemical molecules like proteins and lipids. We chose this code as GROMACS provides extremely high performance compared to all other programs.

In GROMACS, the motion of the atoms is computed using the laws of classical mechanics. GROMACS solves Newton’s equations of motion for a system of $N$ interacting atoms:

$$\frac{d^2 r_i}{dt^2} = \frac{F_i}{m_i}, \quad i = 1, \ldots N$$

where $d^2 r_i/dt^2$ is the acceleration, $F_i$ is the force and $m_i$ the mass.

At each time step, most of the computational cost expense goes into computing the forces on all particles, in particular the non-bonded forces. Their most common form is:

$$V_{nb} = \sum_{i,j} \left[ \frac{1}{4 \pi \varepsilon_0} \frac{q_i q_j}{r_{ij}} + \left( \frac{C_{12}}{r_{ij}^{12}} - \frac{C_6}{r_{ij}^6} \right) \right]$$

The first term contributes to Coulomb interaction while the second contributes to Lennard-Jones interaction. $C_{12}$ and $C_6$ depend on which particle types constitute the pair.

Calculating the non-bonded force normally accounts for 90-95% of the runtime in C/Fortran code. To reduce this cost, we use a cutoff at a chosen distance $r_c$ in the calculation of intermolecular interactions. Only the particles within $r_c$ contribute to the energy of a particle $i$ and they constitute the neighbor list of $i$. A neighbor list is maintained for each particle to speed up force calculation.

Using Merrimac, we simulated a box of 900 water molecules. To get the best performance out of Merrimac we made the following modifications to GROMACS. Each central molecule has a different number of neighbors within its $r_c$. However for performance we want to have neighbor lists of fixed length (for example 16). We loop over all the central molecules $c$ and store their indices in the index array $i\text{central}$ and the corresponding 16 indices of the neighbors in $i\text{neighbor}$. If a certain $c$ has fewer than 16 molecules within the radius $r_c$, dummy molecules are added at the end of its neighbor list. Dummy molecules are given a position far away enough from $c$ so that their interaction with $c$ is negligible. If a certain $c$ has more than 16 neighbors, for example 26, $c$ appears twice in $i\text{central}$ as $c_1$ and $c_2$. $c_1$ has the first 16 neighbors in its neighbor list and $c_2$ has the other 10 neighbor molecules plus 6 dummy molecules. We therefore end up with an array $i\text{neighbor}$ which has a length 16 times the length of $i\text{central}$ (see Table 4.6).

However, this data arrangement is still not sufficient to take advantage of the parallel processing capabilities of each cluster. Merrimac is organized such that each consecutive record in an input stream is sent to the next consecutive cluster. Since we want all clusters to work in parallel without any data dependencies between them, we have to reformat our $i\text{neighbor}$ stream. Instead of storing
4.4 MERRIMAC APPLICATIONS

Original molecule list (No. 900 is the far-away dummy molecule)

\[ \begin{array}{ccccccc}
0 & 1 & 2 & 3 & \ldots & 900 \\
\end{array} \]

Center molecule list: \textit{i\textsubscript{central}} (the superscript numbers indicate how many times the \textit{c} molecule’s index has been repeated.)

\[ \begin{array}{cccccccc}
0^1 & 0^2 & 1^1 & 1^2 & 1^3 & 2^1 & \ldots & 900^a \\
\end{array} \]

Neighbor list: \textit{i\textsubscript{neighbor}} (The subscript numbers indicate the 16 neighbors of each center molecule.)

\[ \begin{array}{cccccccccccccccc}
0^1 & \ldots & 0^1_{16} & 0^2 & \ldots & 0^2_{16} & 1^1 & \ldots & 1^1_{16} & 1^2 & \ldots & 1^2_{16} & 1^3 & \ldots & 1^3_{16} & \ldots \\
\end{array} \]

Table 4.6: Data lists and construction of the neighbor list

Table 4.7 illustrates the final format of \textit{i\textsubscript{neighbor}}. The data are loaded one group after another, going from left to right and then from the top to the bottom, until the end of the neighbor \textit{i\textsubscript{stream}}.

\[ \begin{array}{cccccccccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 \\
0^1 & 0^2 & 1^1 & 1^2 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
0^1 & 0^2 & 1^1 & 1^2 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
0^1 & 0^2 & 1^1_{16} & 1^2_{16} & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\end{array} \]

Table 4.7: Rearranged \textit{i\textsubscript{neighbor}}. Example with 16 clusters.

When the computation is over, the total force on the central molecule is saved to the output force stream, \textit{force\_central}. The output force stream is indexed according to \textit{i\textsubscript{central}}. In this format, central molecules whose index is repeated in \textit{i\textsubscript{central}} because they have more than 16 neighbors are listed multiple times with partial forces only. The final step of the application consists in scatter-adding this partial force stream into a total force stream. This operation was accomplished using the hardware scatter-add support.

**Optimization and Performance on Merrimac**

Optimization of the code occurred in two phases. First, the kernel code dealing with force computation was optimized by use of software pipelining and loop unrolling. Then, the I/O stream operations were optimized via stripmining of the kernel calls.

**Optimization of the Kernel.**

To analyze the effect of optimizations made to the kernel, we rely on the kernel schedules generated by the visualizer tool. Figure 4.28 is the original kernel schedule prior to any optimizations. At this stage, the kernel is roughly 140 cycles long. Prior to the 45th cycle, the functional units
are used only sporadically and with very little operation overlap, despite the fact that the kernel contains 9 independent atom-atom interaction force calculations.

After unrolling the kernel loop and pipelining the code, the kernel schedule improves dramatically, as shown in Figure 4.29. Note that in Figure 4.29, the loop is unrolled twice, meaning that the roughly 165 cycles that make up this schedule represent what would previously have taken two iterations, or roughly 280 cycles, to complete. In Figure 4.29, Numbers along the vertical axis are number of cycles. The various functional units are on the horizontal axis.

Figure 4.28: Kernel without any optimization  Figure 4.29: Kernel after optimization

\textit{I/O Stream Operations:}

Between kernels, the application is always loading the next kernel’s data and saving the previous kernel’s data. The resources used by these I/O stream operations, apart from SRF space, are totally different from the resources used by the computationally intensive kernel. Hence, a perfectly optimized application usually performs almost all its I/O stream operations (e.g. reads/writes between memory and SRF) in parallel with its kernels.

In this application, the I/O stream operations associated with each kernel call are three stream-Gathers to load input and a streamScatterAdd to save output. The scheduler was used to analyse the I/O stream operations. Without any “manual” optimization, the schedule shows double-buffering and indicates that around 150 kernel chunks are called during the application, although all the input data should be able to fit in the SRF in 25 attempts. This inefficiency was removed by manually choosing strip sizes so that all input streams are fed into the SRF evenly. Each kernel chunk then uses all the data in the SRF so that the SRF can be completely cleared and reloaded with new data between kernel chunks. After manual stripmining, the total number of cycles in the application reduces by 13%.

Still, several delays in the I/O stream operations remain. Some load operations that are wholly independent of a running kernel chunk do not occur until the kernel chunk finishes. These delays cause large gaps between kernel chunks. A closer inspection revealed that those delays are due
4.4. MERRIMAC APPLICATIONS

To an insufficient number of MARs (Memory Address Registers). We initially ran the simulator assuming the hardware had only 8 MARs, like the Imagine chip. However, further testing revealed that 22 MARs are needed to run the I/O stream operations without MAR-induced stalling. More importantly, the strip sizes we chose optimally fit the data for one kernel chunk into the SRF, making it near impossible to load any data for the next kernel chunk until the current kernel chunk finished. We addressed this problem by halving the strip-size of the input streams and unrolling the kernel-calling loop twice. The resultant schedule produced after these changes detracted another 240,000 cycles from the application by virtually eliminating the gaps between each kernel chunk. Figure 4.31 is a zoomed in view of the final schedule, placed alongside a zoomed in view of the schedule with 8 MARs (Figure 4.30).

**Figure 4.30: Schedule with 8 MARs**

**Figure 4.31: Schedule with 22 MARs**

**Merrimac vs. Pentium:**

After fully optimizing the application, we found that it will take the Merrimac supercomputer 1.1 million cycles to perform a complete force calculation for a system of 900 water molecules. If Merrimac runs at the projected 1 GHz speed, the force calculation should take roughly 1.1 ms on the actual chip. Meanwhile, a standard Pentium 4 1.7 Ghz processor computer took 62 ms to run the same force calculation in about 105 million cycles. According to the cycle-accurate simulator, the Merrimac chip will compute 22 GFLOPS, which is 39 times more than the Pentium chip could manage at 0.56 GFLOPS.
Bibliography


CITS Roster

Stanford Faculty:
- Alonso, Juan (T,I)
- Dally, Bill (M)
- Darve, Eric (M)
- Durbin, Paul (T)
- Eaton, John (T)
- Fedkiw, Ron (M)
- Ferziger, Joel (T)
- Hanrahan, Pat (M)
- Horowitz, Mark (M)
- Jameson, Antony (T)
- Lele, Sanjiva (T)
- Moin, Parviz (C)
- Pitsch, Heinz (C,I)
- Reynolds, Bill
- Rosenblum, Mendel (M)

External Affiliates:
- Davis, Roger (T)
- Mahesh, Krishman (C)
- Mansour, Nagi
- Wray, Alan (M)
- Barth, Tim (M)
- Constantinescu, George (C)
- Kassinos, Stavros (T)

Stanford Students:
- Ahn, Jung (M)
- Blanquart, Guillaume (C)
- Buck, Ian (M)
- Chelf, Ben (M)
- Das, Abhishek (M)
- Desjardins, Olivier (C)
- Erez, Mattan (M)
- Gopinath, Arathi (T)
- Gummaraju, Jayanth (M)
- Gupta, Amit Kumar (M)
- Hosseini, Kaveh (T)
- Iourokina, Ioulia (T)
- Ihme, Matthais (I)
- Jayasena, Nuwan (M)
- Ji, Minsuk (T)
- Kang, Seongwon (C)
- Kapasi, Ujval (M)
- Knight, Tim (M)
- Kunz, Robert (M)
- Labonte, Francois (M)
- Lantz, Robert (M)
- Laskowski, Greg (T)
- Lee, Ki Hwan (T)
- Losasso, Frank (M)
- Nagarajan, Santhanam (T)
- Pepiot, Perrine (C)
- Sahu, Pradipta (T)
- Shunn, Lee (C)
- Stolte, Christopher (M)
- Tamer, Zaki (T)
- Wall, Cliff (C)
- Zhao, Yanan (M)

Stanford Research Staff:
- Apte, Sourabh (C)
- Ham, Frank (C)
- Herrmann, Marcus (C)
- Fatica, Massimiliano (M)
- Iaccarino, Gianluca (C,T)
- Kalitzin, Georgi (T)
- Kim, Sangho (I)
- Schluter, Jorg (I)
- Van der Weide, Edwin (T)
- Wu, Xiaohua (C,I)

Group legend
C Combustor group
I Integration group
M Merrimac group
T Turbomachinery group