High-throughput Pipelined Mergesort

Kermin Fleming, Myron King, Man Cheuk Ng,
Asif Khan, Muralidaran Vijayaraghavan
MIT - CSAIL
Cambridge, MA
{kfleming,mdk,mcn02,aik,vmurali}@csail.mit.edu

Abstract: *We present an implementation of a high-throughput cryptosorter, capable of sorting an encrypted database of eight megabytes in .15 seconds; 1102 times faster than a software implementation.*

1 Introduction

In the second MEMOCODE hardware/software codesign contest [4] we were given the task of implementing a mechanism to sort an encrypted database. The intended target of the design was specified as a Xilinx XUP development board on which the work can be shared between the PowerPC and the FPGA. In this paper we describe our design methodology and discuss our solution, including some attempts that did not work. The XUP design was completed in approximately three weeks by a three people, while two people worked on a parallel design which was not finished.

With the exception of an AES core, acquired from OpenCores.org[6] and written in Verilog, we implemented our cryptosorter entirely in Bluespec SystemVerilog[1], allowing us to reuse components from our MEMOCODE 2007 design contest submission[5].

The cryptosort task begins with an unsorted encrypted database of 128-bit records placed in external DRAM memory. Initially, the database must be decrypted using a fixed series of keys generated by encrypting the record index with 128-bit AES. After the database has been sorted into ascending order, it must be re-encrypted and placed in memory. Legal database sizes are the powers of two between 2^6 to 2^{18}, inclusive.

2 Design Principles

A typical methodology involves the composition of hardware accelerators with software control. In the case of cryptosort, the control logic is simple enough to implement the entire design in hardware. We decided to develop a single parametric cryptosorter targeting two platforms: the XUP board and the high-performance BEE2 board [2]. We were aware of the risks involved in a full hardware implementation; many of the classical problems of hardware design are exacerbated by a short development window. In particular, system integration becomes challenging due to the volume of new components and the number of implementors. Integration is usually accompanied by rigorous system-level simulation, but, in our case, the run-time and implementation-time overhead precludes this level of verification. To give ourselves a better chance of delivering a working design by the end of the contest, we adopted a set of design principles to help us overcome traditional hardware development risks. First, we required strict adherence to abstract parameterized interfaces. Secondly, we implemented all these interfaces in a latency-insensitive manner, and lastly, to aid in deadlock prevention, we required all requesting modules to reserve space for responses before issuing the request. We believe these principles gave us a competitive advantage in implementing a functioning hardware cryptosorter.

In order to facilitate a parallel multi-platform design effort, we specified a set of abstract parameterized interfaces, which allowed us to work on implementations at various points along the performance-resource utilization curve. Included in these were a set of general peripheral interfaces (e.g. memory) through which we could connect the cryptosorter to platform-specific hardware on target platforms, *without modifying the cryptosorter*. We defined these interfaces in a request-response manner. Our choice of Bluespec SystemVerilog greatly aided this effort because we were able to specify descriptive interfaces and share them among several module implementations.

Latency insensitivity at module boundaries is a convenient way of preventing implementors from making critical timing assumptions about other modules, thereby allowing a system to be composed seamlessly. Along with providing a very useful framework to think about division of tasks, latency insensitivity (in our experience) removes a whole class of integration bugs. Bugs arising from inter-module timing assumptions are particularly difficult to repair because it is difficult to describe and enforce timing assumptions. While adherence to the interfaces is enforced by the compiler, latency insensitivity is much more difficult to enforce, requiring great discipline on the part of the implementor.

As testament to the importance of these principles, our adherence allowed us to entirely avoid writing a system level test bench. Instead, we debugged the system level design on the XUP platform. Most integration problems we encountered were due to incorrect synthesis of our design by the Xilinx toolchain. We did not finish the BEE2 implementation.

3 System Description

Our cryptosorter core, shown in Figure 1, implements the mergesort algorithm. In addition to having optimal $Nlog(N)$ operation-complexity, mergesort has a number of features that make it amenable to hardware implementation. Mergesort parallelizes well; merging independent streams is fully data parallel and the number of streams available to merge scales with the size of the input problem. Since mergesort operates on streams, it is suited to block data

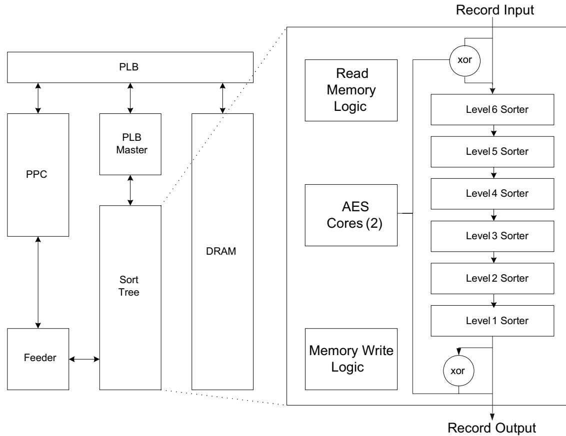


Figure 1. System Architecture

transfers, which are of great practical importance in achieving high memory throughput. Furthermore, a tree of mergers can be constructed to introduce on-chip data locality. Finally, the control logic of mergesort is highly local, allowing for a full-chip implementation without having to deal with the overhead of global communication.

3.1 Control

The three steps involved in sorting an encrypted database are decrypting, sorting, and encrypting. The merging algorithm requires multiple passes over memory, and we were able to combine the decryption stage with the first sorting pass and the encryption stage with the last. Additionally, since the encryption keys are statically known, we generate them at runtime without any additional memory traffic. The control logic makes use of a sort tree, an AES engine, and a memory controller to accomplish this task.

The sort tree merges multiple *sorted* streams into *one* sorted stream. In the first pass over memory, it merges streams of length one, writing streams of length sixty-four to consecutive memory locations. These streams are subsequently merged and at each pass over memory, the length of the sorted output stream grows by a factor of 2^6. When the length of the output stream equals the size of the database, the sort is complete.

The logic which drives the sort tree can be viewed as seventh level of the sort tree (discussed in detail in 3.2). It must first check among the leaf mergers to determine which have room for more data and issues a memory request if space is found. The control issues key requests to the AES engine during the first sort pass to decrypt the records and during the last sort pass to encrypt them. The controller employs a double-buffering scheme in order to maintain memory consistency.

The control logic can issue and multiple memory requests and process multiple responses per cycles, but its throughput is bounded by other modules.

3.2 Sort Tree

The number of memory accesses required to sort a list with a sort tree is $N\lceil log_{2^d}(N)\rceil$, where d corresponds to the depth of the tree. Deeper sort trees, will therefore outperform shallow trees when memory throughput is the system bottleneck. Initially, we explored implementing mergesort with a binary tree, as shown in Figure 2, composed of two-to-one merger modules. Although easy to conceptualize and implement, the area of this structure grows quite quickly as the depth of the sort tree increases. Each merger module requires 3% of slice resources on the XUP, restricting a binary sort tree implementation to a maximum depth of four (fifteen mergers).

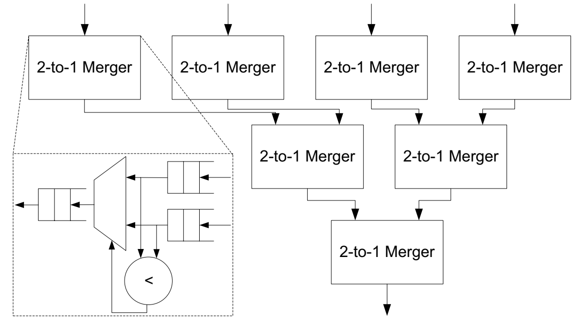


Figure 2. Logical Sort Tree Structure

We observe that the throughput-limiting factor of the sort tree is the root merger, which can output one record per cycle. This bottleneck limits the benefit derived from parallelism in the higher levels of the tree. Consequently, we decided to rate match the sort tree levels by time multiplexing a single comparator among multiple logical mergers at each level. This folding results in the architecture shown in Figure 3, which consists of sort levels in which multiple logical mergers share a single comparator. Since only one merger per level can be active, all the logical mergers in a level can be mapped onto the same dense memory structure, a pair of block RAMs. This mapping not only takes advantage of dense storage resources, but also eliminates the need to use multiplexors to select between mergers. Every cycle, each level's *scheduler* chooses one logical merger for comparison. After comparing records, the smaller of the two records is forwarded to the next level of the tree. Since the levels of the sort tree are very similar, we were able to write a parameterized sort level module that could be to used instantiate all levels of the sort tree.

In order to select a merger for comparison, the scheduler collects the element count for the merger's two source FIFOs, as well as the destination FIFO in the next sort level. If both the source FIFOs contain at least one record, and the destination FIFO is not full, then the merger is ready for execution, and may be scheduled. Since the critical path of the scheduler increases with the number of mergers in the level, some level schedulers did not initially meet timing. To deal with this issue, we created a number of pipelined and combinational scheduler modules, and made the scheduler a parameter of the sort tree level. This allowed us to experiment with a variety of scheduler designs, although ultimately we determined that a simple, greedy scheduler offered satisfactory performance.

The largest sort tree that we were able to fit on the XUP

board had six levels, allowing it to merge sixty-four sorted streams into one sorted stream in one sort pass. With sixty-four to one merging, this sorter requires at most three passes over memory to sort the competition test cases.

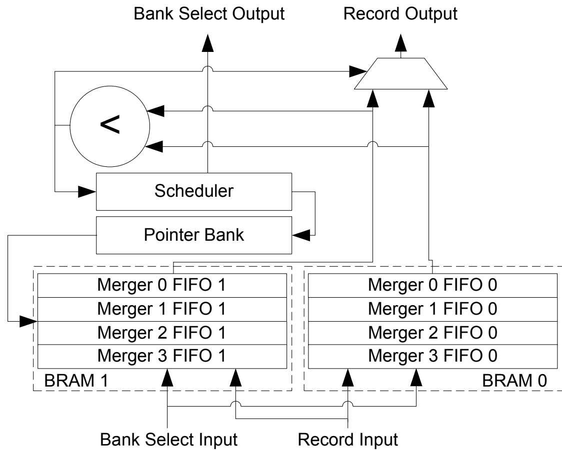


Figure 3. Physical Implementation of Level 3 of the Sort Tree

3.3 AES

The encryption and decryption portion of the cryptosorter task requires an implementation of the AES[3] algorithm. We acquired an existing AES core from OpenCores.org[6]. This core can perform a complete encryption sequence in twelve clock cycles: one cycle for key expansion at the beginning, ten cycles for performing the ten encryption rounds, and one cycle for buffering the output.

Due to the resource limitations of the XUP board, the XUP cryptosorter instantiates only two AES cores, giving a throughput of $\frac{1}{6}$ of an encryption value per cycle. We anticipated that the throughput of the AES engine would be a system bottleneck. To alleviate this issue, the AES cores were modified to run in a separate clock domain. In the final XUP implementation, the AES cores were clocked at 120 MHz, which increases the aggregate throughput of the cores to $\frac{1}{5}$ of an encryption value per cycle, which resulted in a small performance increase on the two benchmarks, shown in Figure 5(b).

3.4 Memory Interface

The cryptosorter interfaces to the system memory through a high-speed PLB bus master, a core originally developed for the MEMOCODE 2007 design contest. The bus master supports burst requests of sixteen sixty-four bit beats and may have a load and a store request in flight simultaneously. The bus master supports a relaxed memory ordering, allowing loads and stores to bypass one another.

Our testing shows that the setup time for a burst transfer is approximately 14 cycles, after which data beats proceed on back to back cycles. The default Xilinx DDR controller supports only a single 64-bit load or store transfer per cycle. Therefore, since we issue sixteen beat requests, we expect

Problem Size	Single 100MHz Domain		120MHz AES Domain	
	Random (μs)	Rotated (μs)	Random (μs)	Rotated (μs)
2^6	8.8	8.8	7.8	7.7
2^{10}	165.1	166.0	160.9	161.7
2^{14}	3756.0	3740.3	3744.8	3732.8
2^{18}	59604.2	59491.1	59657.1	59612.4

Figure 4. Benchmark Results for Cryptosorter

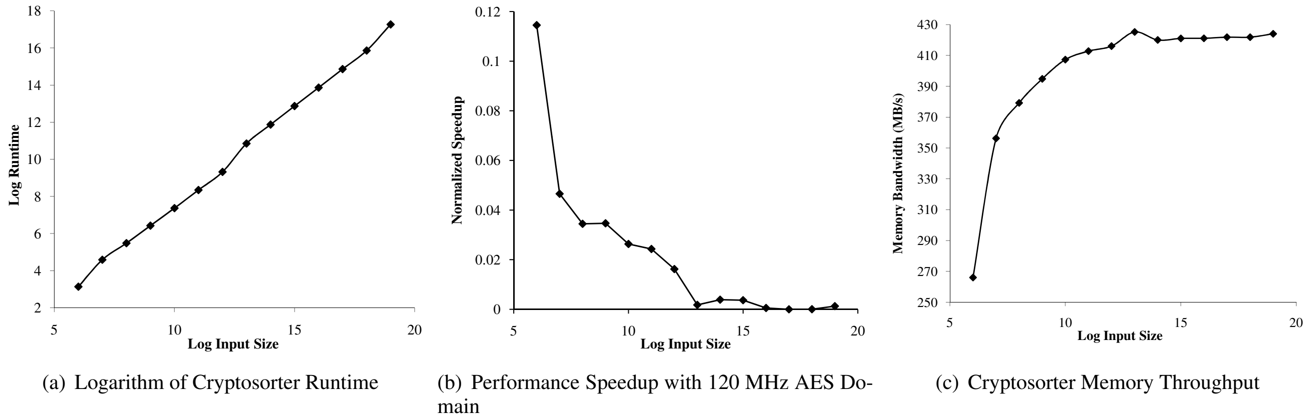
to achieve no more than 427 MB/s of memory bandwidth, approximately one record every four cycles. Since every record passed through the cryptosorter must be loaded and stored, the effective cryptosorter throughput can be no more than record every eight cycles.

4 Evaluation

Since each pass of the cryptosorter merges up to sixty-four sorted streams into a single sorted stream, the algorithmic time complexity of the entire procedure is $KN\lceil log_{2^6}(N)\rceil$, where N is the size of the input and K is some constant. This bound implies that in most cases, doubling the input size will double the processing time, except when the input size crosses a power of 2^6 boundary. In this case, the processing time should more than double due to the need for another pass over the database. This effect is illustrated in Figure 5(a) by the slight kink between the 2^{12} and 2^{13} input sizes. Figure 4 contains raw runtime results for the input sizes used to compute the design contest performance metric, an equal weight normalized geometric mean. The random benchmark sorts a random database, while the rotated benchmark sorts a partially sorted database. Our implementation is insensitive to the order of input data. Therefore, the speedups we achieve are the same for the two benchmarks provided. According to the metric, our cryptosorter achieved a factor of 1102 performance increase over the reference software implementation. At peak performance, we achieve approximately 0.8 comparisons per cycle at a memory throughput of 424 MB/s.

For most input sizes, runtimes can be predicted using the previously mentioned bound with K equal to .075 us, the minimum memory round trip time for a single record. The smaller input sizes, however, diverge from this theoretical model. To explain these empirical results, we examine the modular throughput analysis contained in section 3. The maximum memory bandwidth, roughly one record every four cycles, is insufficient to saturate either the control logic or the sort tree, both of which are capable of achieving throughputs of at least one record every two cycles. Analyzing the interaction of the AES engine and the memory controller will help us understand the throughput of the cryptosorter in these deviant cases.

The AES engine, even when clocked to 120MHz, produces one AES key every five cycles, while the memory



(a) Logarithm of Cryptosorter Runtime

(b) Performance Speedup with 120 MHz AES Domain

(c) Cryptosorter Memory Throughput

Figure 5. Cryptosort Runtime and Performance Metrics

returns a record once every four cycles. At first glance, it would seem that these mismatched rates would reduce the observed memory bandwidth to 80% of the theoretical maximum (340 MB/s). However, the observed bandwidth, shown in Figure 5(c), exceeds this percentage substantially, except in the case of the smallest input. Figure 5(b) suggests that, except in the case of the the smaller inputs, upclocking the AES provides no performance benefit. For inputs larger than 2^{12}, a portion of the memory bandwidth increase is due to the middle sorting passes over memory, which do not involve the AES engine. The remainder of the bandwidth differential can be attributed to the relaxed memory ordering used by the PLB Master. For input sizes larger than 2^6, the first pass over memory (in which the initial data is decrypted) is able to hide the latency of the AES engine by overlapping load/decrypt operations with store operations. In the last pass, the latency of the store/encrypt operation is similarly hidden by overlapped loads. For smaller input sizes, the cost of filling and draining the sort tree is not amortized; in addition, the absence of overlapping opportunities exposes the AES-bottleneck.

The synthesis results of the final design are included in Figure 6. Unsurprisingly, the sort tree accounts for most of the resource utilization. Slice usage of sort tree levels two through six scale approximately linearly with the number of logical mergers present. Sort tree level one does not scale, since its single internal merger is implemented with registers rather than BRAMs. The control logic requires almost as many resources as the sixth level of the sort tree, since it must maintain roughly the same number of pointers.

The sorter core requires 55% of the slices and 47% of the BRAMs available on the Virtex-II Pro 30 FPGA, while the full system requires 73% of slices and 58% of the BRAMs.

There are several modifications which we believe will further increase the performance of the cryptosorter. Expanding the AES engine to three cores would improve sort times for smaller input sizes. The addition of a a seventh sort tree level would improve sort times for most input sizes. However, a seventh sort level would likely not fit on the XUP since it requires an additional 25% of the slices. Increasing the memory burst size beyond sixteen is the simplest and perhaps most beneficial of the potential improvements. If large bursts maintain the same back-to-back beat transfer of the size sixteen burst, then such a change would dramatically increase the theoretical maximum memory bandwidth. Such a change would have limited impact on area, perhaps only increasing the number of BRAMs consumed by the sort tree.

Module	Total Slices	LUTs	Flip Flops	BRAMs
Control	1586	585	2898	4
PLB Master	726	749	1359	0
AES Engine	1024	947	1879	20
AES Core	430	405	804	10
Sort Tree	4096	1941	7748	40
Sort Level 6	1875	706	3583	8
Sort Level 5	722	235	1345	8
Sort Level 4	434	117	799	8
Sort Level 3	355	65	653	8
Sort Level 2	271	36	492	8
Sort Level 1	916	784	1699	0
Sorter Total	7658	4487	14294	64
System Total	10051	6367	17069	80

Figure 6. Synthesis Results for Cryptosorter

References

[1] Bluespec Inc. http://www.bluespec.com.

[2] C. Chang, J. Wawrzynek, and R. W. Brodersen. Bee2: A high-end reconfigurable computing system. *IEEE Des. Test*, 22(2):114–125, 2005.

[3] J. Daemen and V. Rijmen. Rijndael for aes. In *AES Candidate Conference*, pages 343–348, 2000.

[4] Krste Asanovic and James Hoe and Patrick Schaumont. The Second MEMOCODE Design Contest. `http://rijndael.ece.vt.edu/memocontest08`, March 2008.

[5] N. Dave, K. Fleming, M. King, M Pellauer, M. Vijayaraghavan. Hardware Acceleration of Matrix Multiplication on a Xilinx FPGA. In *Proceedings of Formal Methods and Models for Codesign (MEMOCODE)*, Nice, France, 2007.

[6] Rudolf Usselmann. `http://www.opencores.org/cvsweb.shtml/aes_core`.