Superoptimized Memory Subsystems for Streaming Applications

Joseph G. Wingbermuehle
Ron K. Cytron
Roger D. Chamberlain

Dept. of Computer Science and Engineering
Washington University in St. Louis

Superoptimized Memory Subsystems for Streaming Applications

Joseph G. Wingbermuehle, Ron K. Cytron, and Roger D. Chamberlain
Dept. of Computer Science and Engineering
Washington University in St. Louis
{wingbej,cytron,roger}@wustl.edu

ABSTRACT

Because main memory is many times slower than modern processor cores, deep, multi-level cache hierarchies are ubiquitous in computers today. Similarly, applications deployed on ASICs and FPGAs are often hindered by slow external memories. Therefore, to achieve good performance, hardware designers must optimize main memory usage. Unfortunately, this process is often labor intensive and fails to explore the full range of potential memory designs. To address this issue for applications expressed in a streaming manner, we show that it is possible to generate automatically a *superoptimized* memory subsystem that can be deployed on an FPGA such that it performs better than a general-purpose memory subsystem. Rather than explore only simple memory subsystems, our superoptimizer is capable of exploring extremely complex designs consisting of multi-level caches and other components. Finally, we show that it is possible to deploy applications with superoptimized memory subsystems with minimal additional effort while achieving significant performance improvements over a naive memory subsystem.

Categories and Subject Descriptors

B.6.3 [**Logic Design**]: Design Aids; B.3.2 [**Memory Structures**]: Design Styles

1. INTRODUCTION

Due to the large disparity in performance between main memory and processor cores, large cache hierarchies are a necessary feature of modern computer systems. By exploiting locality in memory references, these cache hierarchies attempt to reduce the amount of time an application spends waiting on memory accesses. Although cache hierarchies are most common, one can extend this notion to a generalized on-chip memory subsystem, which acts as an interface between the computation elements and off-chip main memory.

For general-purpose computers, memory subsystems are designed to have the best performance across a large range of applications. However, due to the general-purpose nature of these memory subsystems, such a memory subsystem may not be optimal for a particular application. Because of the potential performance benefit with a custom memory subsystem, we propose the use of memory subsystems tailored to a particular application. Such custom memory subsystems have been used for years for applications deployed on ASICs and FPGAs [3,10,19]. Further, it is conceivable that general-purpose computer systems may one day be equipped with a more configurable memory subsystem if the configurability provided enough of a performance advantage.

Although selecting the parameters for custom cache hierarchies optimally is an active area of research, most previous work focuses on a fixed topology. Recently, superoptimization has been employed to expand the search space for memory subsystems to contain multiple memory subsystem components such as caches, scratchpads, and address transformations, allowing for these components to be combined arbitrarily [27]. However, that work only considers single-threaded applications. Here we extend that work by considering custom memory subsystems for pipelined, streaming applications.

Streaming is a parallel programming paradigm in which application kernels communicate over fixed communication channels. The streaming paradigm is used in systems such as ScalaPipe [26] and StreamIt [23], among many others [4,21]. Within the streaming paradigm, conceptually, each kernel has its own independent memory address space. Communication between kernels is performed via communication channels implemented as FIFO buffers. Unlike a single-threaded application, which has a single memory subsystem to optimize, a streaming application can potentially have a separate memory subsystem for each kernel. In addition, each communication channel or FIFO between kernels is yet another memory subsystem to be optimized.

The concept of superoptimization was introduced with the goal of finding the smallest instruction sequence to implement a function [15]. This differs from traditional program optimization in that superoptimization attempts to find the best sequence at the expense of a potentially long search process rather than simply improving code. In a similar vein, we are interested in finding the best memory subsystem at the expense of a potentially long search process rather than a generic memory subsystem.

Traditionally, superoptimizers have used exhaustive search; however, exhaustively searching for the best memory subsystem would be prohibitively time-consuming. This is because in evaluating a particular memory subsystem, we simulate an address trace from the application. Due to the size of these traces, the simulation time can be extremely time consuming. Therefore, as in [20] and [27], we use a stochastic search technique instead of exhaustive search.

Due to the number of memory subsystems in a streaming application, the already complex problem of superoptimizing a single-threaded address trace is compounded. This is because, in addition to a shared resource constraint, the performance of one kernel can affect another both directly, by moving the bottleneck, and indirectly, by consuming excessive main memory bandwidth. Thus, we use a heuristic to guide the search to those memory subsystems that are most likely to benefit the application.

To evaluate our superoptimized memory designs, we target an FPGA with an external LPDDR main memory. The FPGA device is a Xilinx Spartan-6 LX45 clocked at 100 MHz. The external LPDDR is a 512 Mib device clocked at 100 MHz. All memory subsystems share access to the external LPDDR memory device. The Spartan-6 LX45 has 116 block RAMs (BRAMs), which we use to implement our custom memory subsystems. Each BRAM is 18 Kib, providing a total of 2,088 Kib on-chip memory.

By evaluating our applications on a physical device, we show that it is possible to achieve real performance improvements over a generic memory subsystem with minimal extra effort.

2. RELATED WORK

Here we extend the work on superoptimized memory subsystems for single-threaded applications [27] by considering streaming applications. Further, we show actual performance improvements for applications implemented on an FPGA device in addition to simulated reductions in execution time.

There is related work in making off-chip memory easier to use. LEAP scratchpads [1] provide a portable memory abstraction for hardware kernels. These memory abstractions may contain caching and can be backed by a larger main memory. CoRAM [6] is a technology similar to LEAP scratchpads, but lower-level, that provides an SRAM-style interface to memory. Unlike block RAM resources embedded in the FPGA, however, CoRAMs can be backed by a larger main memory. Both LEAP scratchpads and CoRAMs provide a similar function to our memory subsystem generator. However, unlike these works, we are concerned with automatic discovery of memory subsystems rather than an explicit description.

Prefetching [29] has been considered to improve the performance of applications deployed on an FPGA when combined with a memory abstraction such as LEAP scratchpads or CoRAM. Unlike our work, that prefetching mechanism is generic and dynamically discovers strided accesses.

MPack [24] attempts to optimize the packing of data into block RAM resources. In our work we do not consider packing multiple subsystem components, though doing so could allow for a higher utilization of block RAM resources.

Another source of related work comes from the automatic selection of cache parameters. Such optimization has been demonstrated for single-level caches [9] and two-level caches [10,11]. In addition, there have been approaches to changing certain cache parameters dynamically [2, 22, 25]. However, these techniques do not consider the potential for a more complex memory subsystem.

An approach to optimizing both the computation and communication between kernels in streaming applications is presented in [7]. In [28], an approach to improving the memory behavior for FPGA applications implemented in a high-level language such as C or C++ is presented. Unlike these works, we treat the computation as fixed, but consider a wider search space for memory subsystems.

Finally, non-traditional memory subsystems, such as victim caches [13], combinations of caches and scratchpads [18, 19], and split caches [17] have been considered. However, these techniques represent specific modifications to the memory subsystem that are intended to be generic rather than customized for a particular application.

3. METHOD

Given an application to be deployed on either an ASIC or FPGA, the process to create a custom memory subsystem consists of several steps. First, the design without a memory subsystem is evaluated to determine what ASIC or FPGA resources are not used and, therefore, available for the memory subsystem. Next, an address trace is gathered for a single run of the application. This address trace is then fed into the memory subsystem superoptimizer, which proposes memory subsystems and simulates them to determine their performance. Finally, the memory subsystem generator is used to generate a custom HDL design for the application to use.

3.1 Address Traces

To superoptimize a memory subsystem, we first require an address trace. Unlike single-threaded address traces, we require a separate trace per kernel (note that each kernel has its own memory subsystem). In addition, communication between kernels must be recorded to allow us to model accurately the parallel kernels and optimize the size of the FIFOs between the kernels. Finally, some notion of the computation time between memory accesses must be recorded to predict accurately how long each kernel will run relative to other kernels.

$$\begin{tikzcd}
 & A \arrow[dl, "1"'] \arrow[dr, "2"] & \\
B \arrow[dr, "3"'] & & C \arrow[dl, "4"] \\
 & D &
\end{tikzcd}$$

Figure 1: Split-Join Topology

Consider the simple streaming application topology shown in Figure 1. The vertices of the graph represent kernels and the edges represent communication channels. Here we have four kernels (`A`, `B`, `C`, and `D`) where kernel `A` produces data on two channels (1 and 2) and kernel `D` consumes data on two channels (3 and 4).

Each kernel has a separate address trace. For example, the trace for kernel `B` might look something like:

Component	Description	Parameters ($n \in \mathbb{Z}_+$)	Latency (cycles)
Cache	Parameterizable cache	Line size (2^n) Line count (2^n) Associativity ($1 \ldots line\_count$) Replacement (LRU, MRU, FIFO, PLRU) Write policy (write-back, write-through)	3
FIFO	FIFO implemented in BRAM	Depth (2^n)	1
Offset	Address offset	Value ($\pm n$)	0
Prefetch	Stride prefetcher	Stride ($\pm n$)	0
Rotate	Rotate address transform	Value ($\pm n$)	0
Scratchpad	Scratchpad memory	Size (2^n)	2
Split	Split memory	Location (n)	0
XOR	XOR address transform	Value (n)	0

Table 1: Memory Subsystem Components

```
Consume an element from channel 1
Read 4 bytes from address 0x1234
Perform a computation taking 8 cycles
Write 8 bytes to address 0x200
Produce an element on channel 3
```

Recording the interaction over the communication channels as *produce* and *consume* allows the superoptimizer to size the the FIFOs used for the communication channels without affecting correctness of the application, provided the FIFOs are at least as large as the application requires.

Although recording an address trace for *split* kernels (such as kernel `A` in Figure 1) and *join* kernels (such as kernel `D`) would provide a valid trace, such a trace may not give the superoptimizer sufficient freedom to size the FIFOs. For example, if kernel `A` were a load balancer, it might output more items to one channel than the other depending on its ability to write to the channel. To handle such situations, our simulator is capable of modeling certain *split* and *join* kernels intrinsically without using an address trace.

There are several ways to obtain address traces. Because our benchmarks are implemented in ScalaPipe, we modified ScalaPipe to have the ability to dump an address trace for kernels mapped to processor cores. This allows us to run the application first on general-purpose processor cores to gather the address traces. After an address trace is gathered, the application can be mapped to an FPGA device for deployment.

An additional benefit to using ScalaPipe to gather the traces is that, since ScalaPipe is capable of high-level synthesis, we can also record the number of cycles that the computation will take between memory accesses in the address trace. This information allows the superoptimizer to divide the memory resources among the kernels more effectively.

For benchmarks not implemented in ScalaPipe, it is possible to instrument the application manually to generate the required trace data. For example, an application implemented in a hardware description language (HDL) could be manually instrumented and then run in a simulator.

3.2 Simulation

To evaluate the performance of the memory subsystems proposed by the superoptimizer, we use a custom trace-based memory simulator. We use a custom memory subsystem simulator for three reasons. First, we need to simulate complex memory subsystems beyond simple caches. Second, rather than the number of cache misses, we are interested in the total run time of the application: cache misses alone would not provide enough information to the superoptimizer for it to decide between a single level and a multi-level cache and memory access time is insufficient for deciding how to divide up the memory resources among multiple memory subsystems. Finally, the simulator must be fast enough to simulate large traces for many thousands of repetitions in a reasonable amount of time.

Parameter	Description	Value
Frequency	DRAM I/O frequency	100 MHz
CAS	Cycles to select a column	3
RCD	Cycles from open to access	3
RP	Cycles required for precharge	3
Page size	Size of a page in bytes	1024
Page count	Number of pages per bank	8192
Width	Channel width in bytes	2
Burst size	Number of columns per access	8
Page mode	Open or closed page mode	closed
DDR	Double data rate	true

Table 2: Main Memory Parameters

For the results presented here, our simulator accommodates the memory subsystem components shown in Table 1. The simulator reports the total cycles that the application would take to run on our target platform. For the main memory, the simulator assumes that there is a priority arbiter in front of the main memory with a single read/write port. The main memory is modeled as a DRAM device with the parameters shown in Table 2, which were chosen to closely model our experimental platform.

3.3 Optimization

We use *old bachelor acceptance* [12] to explore the search space of memory subsystems. Old bachelor acceptance extends threshold acceptance [8], which is a stochastic hill-climbing technique similar to simulated annealing [14]. Old bachelor acceptance provides a compromise between search space exploration and hill climbing. Thus, although we may not obtain the best possible memory subsystem with this technique, we do see fairly good results in much less time than it would take to perform an exhaustive search.

As in [27], a proposal for a particular memory subsystem involves (1) the insertion of a memory subsystem component, (2) the removal of a component, or (3) a change to one of the component's parameters. This allows the generation of arbitrarily complex memory subsystems. The components supported by the superoptimizer are shown in Table 1.

The search space is much larger when multiple kernels are considered than it is for single-threaded applications. Because of this, in addition to the address-selection heuristic presented in [27], our superoptimizer employs a heuristic to guide it to spend more effort exploring the memory subsystems that are most likely to benefit the application. To do this, the subsystems for each kernel and FIFO are weighted by the product of their resource usage and their total memory access time. The superoptimizer then randomly selects a subsystem to modify based on these weights. This causes the superoptimizer to spend more time on those memory subsystems that consume a large portion of the resources and those with the most room for improvement.

Since our target device is an FPGA, we constrain the superoptimization process by FPGA resources. Specifically, we constrain the superoptimization process such that the final application uses no more than 80% of the slices and no more than 80% of the BRAMs available on the FPGA. By constraining the resources to 80%, we prevent the design from becoming too congested, which could prevent the design from being routed or meeting timing closure. In addition to the resource constraints, we place a lower bound of 100 MHz on the system clock for the design. The clock constraint prevents the superoptimizer from slowing down the computation with an overly-complex memory subsystem.

To enforce the resource constraints, rather than build each proposed memory subsystem, the superoptimizer tracks the resource usage of each memory subsystem component by storing synthesis results in a database. The sum of the resources used for each component are then used in the superoptimization process. To ensure the design will run at the required frequency, memory subsystem components whose synthesis estimates are less than 100 MHz are discarded.

Although the constraints on BRAMs and slices are fairly conservative, the constraint on frequency could easily be broken with too complex of a design. To address this, we maintain an estimate of the maximum path length in the superoptimizer and use the estimate as an additional constraint.

3.4 Subsystem Generation

Once a memory subsystem has been superoptimized, we use an automatic memory subsystem generator to generate a VHDL description of the memory subsystem. This subsystem generator is capable of generating all of the memory subsystems shown in Table 1. Each subsystem has a simple SRAM-style interface with per-byte write enables. The subsystems are connected to the main memory using a priority arbiter capable of allowing multiple outstanding main memory requests (one for each subsystem).

The word size of various components in the memory subsystem can differ. To handle this, an adapter is inserted between each component in the memory subsystem. For example, if there is a two-level cache where the first level has a word size of 8 bytes and the second level has a word size of 16 bytes, the adapter will direct the reads to the correct part of the larger word and set the byte mask appropriately for writes. If the second level cache has the smaller word size, each access from the first level cache will be turned into multiple accesses. For simplicity, the word size is restricted to a power of two.

4. BENCHMARKS

Following is a description of the benchmarks used to evaluate our custom memory subsystems. All of the benchmarks are implemented in ScalaPipe [26]. ScalaPipe is a streaming application generator that allows one to author an application in a high-level language and then generate code for deployment on traditional processors and FPGAs.

We have enhanced ScalaPipe with the ability to generate applications that output memory address traces for kernels deployed on standard processor cores and to use our custom memory subsystems for kernels deployed on FPGAs. This allows us to deploy the application first on processor cores to generate the address traces and then generate the application on FPGA cores for deployment with our custom memory subsystems.

4.1 Merge Sort

The first application we consider is a merge sort capable of sorting up to one million 32-bit integers [5]. This application makes use of a generic merge kernel with a single input channel and a single output channel. The kernel is replicated $\lceil \lg n \rceil$ times to sort n elements, as shown in Figure 2. Each kernel in the pipeline sorts sequences of elements $2\times$ longer than the sequences from the preceding kernel by using an internal buffer to store half the elements.

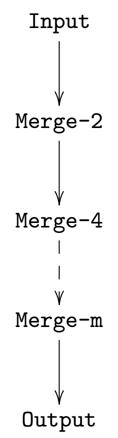

Figure 2: Merge Application Topology

Due to the memory requirements of sorting one million integers, this application requires off-chip memory. However, exactly how the BRAM resources of the FPGA should be divided up among the kernels and FIFOs is not immediately apparent and is the subject of investigation.

4.2 n-Body

The next benchmark we consider is an application to simulate the 3-dimensional n-body problem using the naive $O(n^2)$ algorithm. An n-body simulation predicts the positions and velocities of point masses in space at various times. The naive algorithm updates each point by considering the gravitational effect of all other points. The topology of the n-body application is shown in Figure 3.

In the n-body application, the `Input` kernel reads the initial positions of each particle to be simulated. The `Buffer` kernel buffers the points for the next iteration (or from input on the first iteration). Next, the `Streamer` kernel sends the particles past the `Force` kernel, which computes the forces on

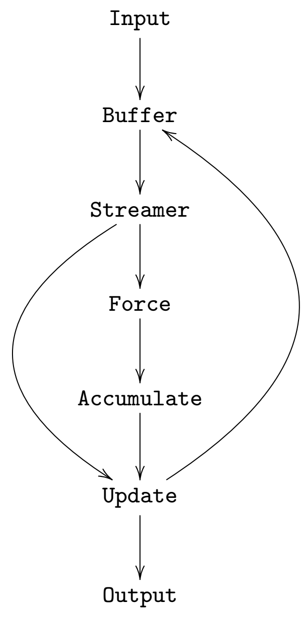

Figure 3: n-body Application Topology

each particle. The `Accumulate` kernel then sums the forces on each particle. Once the total force on a particle has been computed, the `Update` kernel updates the particle's position and velocity, sending the results to both the `Output` and `Buffer` kernels. Finally, the `Output` kernel saves the results.

In this application, there are two kernels that use off-chip memory: the `Buffer` kernel and the `Streamer` kernel. In addition to the memory subsystems used by these two kernels, there are eight FIFOs to be optimized. Although it would be possible to simulate a small number of particles without using off-chip memory, larger problems necessitate the use of off-chip memory, leaving us to determine how to best use the BRAM resources.

4.3 Laplace

The Laplace benchmark is an application to solve Laplace's equation using a Monte-Carlo technique [26]. Laplace's equation is a partial differential equation that can be used to model steady-state heat diffusion. The topology of this application is shown in Figure 4.

In the Laplace application, random numbers are generated using the Mersenne twister [16] random number generator in the `RNG` kernel. The `Split` kernel divides the random numbers among two `Walk` kernels, which perform a random walks from each position of interest. Next, the `Avg` kernel averages the results of the random walks and sends the output to the `Output` kernel.

The only kernel in this application to use a memory array is the `RNG` kernel, which uses 2,496 bytes of memory. So although this application does not require the use of off-chip memory, off-chip memory could potentially be used effectively for the `RNG` kernel and one or more of the FIFOs.

4.4 Matrix-Matrix Multiply

The matrix-matrix multiply benchmark is a streaming application to perform matrix-matrix multiplication on two 256x256 matrices of 32-bit floats. The topology is shown in Figure 5.

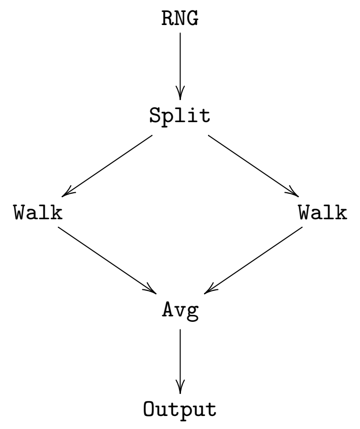

Figure 4: Laplace Application Topology

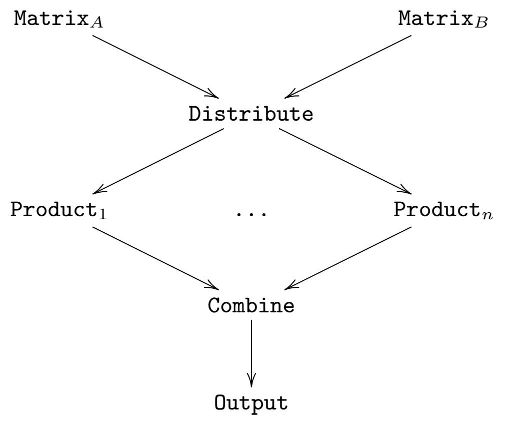

Figure 5: Matrix-Matrix Multiplication Topology

In the matrix-matrix multiply benchmark, the source matrices are provided by the `Matrix`$_A$ and `Matrix`$_B$ kernels. The `Distribute` kernel holds the matrix data and streams it past the `Product` kernels. Each `Product` kernel performs a dot product. In our experiments, we use two `Product` kernels. Next, the `Product` kernels send the dot products to the `Combine` kernel, which collects the results in the correct order. Finally, the `Output` kernel outputs the results.

With this benchmark, only the `Distribute` kernel uses a memory array: one to store the matrices totaling 524,288 bytes. In addition to this memory subsystem, there are FIFOs connecting all of the kernels, which could potentially be resized.

4.5 Median

Finally, we consider an application to find the median of a stream of up to one million unique integers. This benchmark is a simple two-stage pipeline, shown in Figure 6, where the `Hash` stage removes duplicates using an open-address hash table and the `Heap` stage uses a binary heap to recover the median value.

In the median application, both the `Hash` and the `Heap` kernels require more memory the FPGA has available. The `Hash` kernel uses 8 MiB and the heap kernel uses 4 MiB. In addition to the memory subsystems for the kernels, the

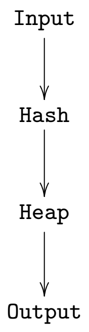

Figure 6: Median Application Topology

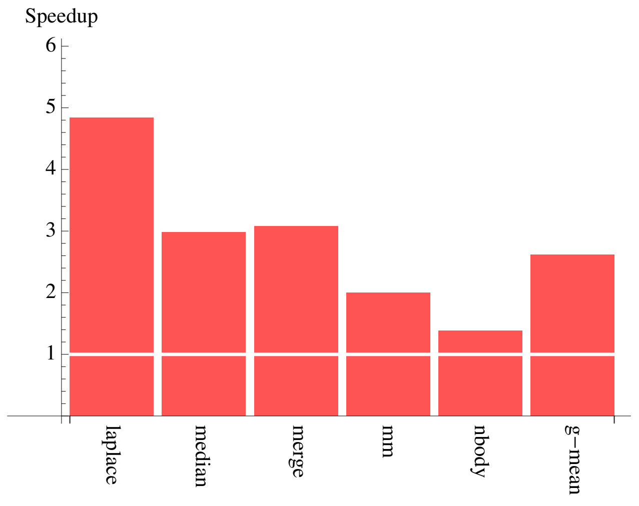

Figure 7: Simulated Speedup

FIFO between the kernels is another subsystem whose size and implementation is to be optimized.

5. RESULTS

Our results are presented with respect to the following baseline: we implement all FIFOs as registers (FIFOs that can hold a single element). All memory subsystems for kernels are connected directly to the arbiter for the main memory. This type of memory structure uses the least amount of area on the FPGA device and requires the least amount of effort to implement. Thus, although it might not be the final design for a particular application, it does represent a likely starting point.

Figure 7 shows the speedup of the superoptimized memory subsystems over the baseline for each benchmark as reported by the memory simulator (the *g-mean* bar shows the geometric mean). Because the simulator takes into account computation time as well as memory access time, the simulated speedup should be an accurate representation of the actual speedup one would expect to obtain by running the application on the physical device. However, there are two potential sources of error. The first is the main memory model, which does not take all possible parameters into account (for example, refresh is not included in the simulated memory model). The other source of error is the application input and output, which is done over a USB interface.

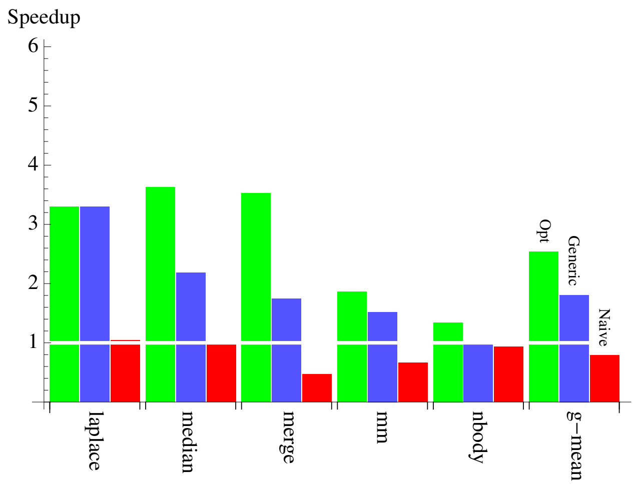

Figure 8: Actual Speedup

Figure 8 shows the actual speedup from running each benchmark on the FPGA device described in Section 1. The first bar in each group shows the speedup over the baseline for the superoptimized memory subsystem. As before, the *g-mean* group shows the geometric mean.

In addition to a comparison of the superoptimized memory subsystem against the baseline, we also compare a generic memory subsystem as well as a naive memory subsystem to the same baseline. For the *generic* memory subsystem (the second bar in each group in Figure 8), each kernel memory subsystem has a 8 KiB direct-mapped cache and each FIFO is 256 items deep and implemented in BRAM. This generic memory subsystem demonstrates the performance one might expect from a memory subsystem that was selected without considering the implementation details of the kernels.

For the *naive* memory subsystem (the last bar in each group in Figure 8), no BRAM is used for the kernel memory subsystems and each FIFO is 256 items deep implemented in main memory instead of BRAM. The naive memory subsystem attempts to demonstrate a worst-case memory subsystem where every access contends for main memory.

Comparing the actual results to the simulated results, we see that in most cases the actual speedup was slightly higher than the simulated speedup. This is due to the fact that reducing the number of main memory accesses improves performance more than the simulated memory model predicts. However, for the Laplace benchmark, the actual speedup is less than predicted. Again, this is due to the main memory model since, as we will see later, two of the FIFOs between kernels were moved into main memory rather than using BRAMs.

5.1 Laplace

The Laplace benchmark exhibits a smaller speedup than the simulation would imply. For the Laplace benchmark, the superoptimizer selected a 4,096-byte scratchpad for the `RNG` kernel. This moves all memory accesses `RNG` into the faster BRAM, avoiding the main memory completely. In addition, several of the FIFO sizes were adjusted, as shown in Table 3.

Because the superoptimizer tries to find the memory subsystem that provides the lowest execution time using as few resources as possible, several of the FIFOs are implemented

FIFO	Depth	Implementation
RNG → Split	1	register
Split → $\texttt{Walk}_1$	256	main memory
Split → $\texttt{Walk}_2$	256	BRAM
$\texttt{Walk}_1$ → Avg	64	main memory
$\texttt{Walk}_2$ → Avg	8	BRAM
Avg → Output	1	register

Table 3: Laplace FIFO Implementations

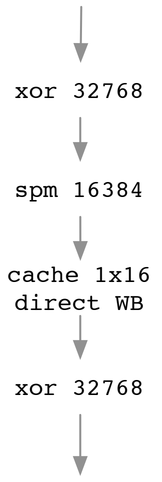

Figure 9: Subsystem for the Hash kernel

in main memory rather than directly in BRAM. According to the simulation model, this does not slow down the benchmark since the computation time is able to hide the memory latency. However, since the main memory model is imprecise, there is a benefit to implementing the FIFOs in BRAM that is unknown to the superoptimizer. By implementing all of the FIFOs in BRAM, we are able to obtain a speedup slightly better than the simulation model estimates.

For the Laplace benchmark, the superoptimized memory subsystem provides more than a 3× speedup over the baseline. However, the generic memory subsystem provides a similar speedup. This is because this benchmark is very sensitive to the size of the FIFOs. Thus, even the naive memory subsystem offers a performance improvement over the baseline due to the increased FIFO sizes.

5.2 Median

For the median benchmark, the superoptimized memory subsystem for the Hash kernel is shown in Figure 9. In the figure, memory accesses from the kernel enter the top and memory accesses to the main memory exit the bottom. In this particular memory subsystem, the address is transformed by flipping a bit (xor). The address transformation is followed by a 16,384-byte scratchpad, which is followed by a single-entry cache having a single line that is 16 bytes (the WB in Figure 9 stands for write-back). Finally, the last address transformation reverses the first transformation. Note that the superoptimizer automatically inserts address transformations in pairs like this to ensure the correct section of main memory is accessed.

The effect of the address transformation is to move certain parts of the hash table into the scratchpad. The cache can

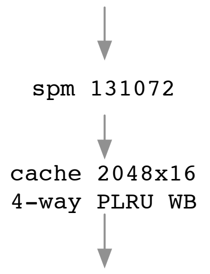

Figure 10: Subsystem for the Heap kernel

be helpful here since the main memory interface is 16-bytes wide and we only access 4 bytes at a time. Therefore, the cache allows us to avoid main memory accesses when multiple words are requested within the same 16-byte range.

The superoptimized memory subsystem for the Heap kernel is shown in Figure 10. Again, we have a scratchpad followed by a cache. This is logical for a binary heap structure since the early addresses are accessed much more frequently than later addresses.

Finally, the FIFO between the Hash and Heap kernels is 16 entries deep and implemented in BRAM. This allows the Hash kernel to keep running even if the Heap kernel backs up. The other FIFOs are 1 entry deep.

5.3 Merge Sort

The merge benchmark has 15 memory subsystems for the Merge kernels and 23 memory subsystems for FIFOs, giving a total of 38 memory subsystems. Although there are 20 Merge kernels, only 15 have memory subsystems since ScalaPipe does not generate memory subsystems if the size of the memory is less than 1,024 bytes.

For the Merge kernels with smaller memory subsystems that need to store fewer than 32,768 bytes, the superoptimizer selects scratchpads. However, for the larger memory subsystems, the superoptimizer selects small, direct-mapped caches. The scratchpads allow the smaller memory subsystems to run without accessing main memory at all. The small direct-mapped caches, on the other hand, reduce the number of accesses going to main memory since the main memory is 16 bytes wide and each access is only 4 bytes.

Most of the FIFOs between kernels were selected to be a single element deep and implemented as a register. However, several of the FIFOs between the later stages are 1,024 and 2,048 elements deep implemented in BRAM. This is because the access latency between the later stages will vary since not all the accesses will hit in cache.

In terms of performance, the superoptimized memory subsystem for the merge sort benchmark is over 3× the baseline memory subsystem and closely matches what the simulation predicted. In this case, the generic memory subsystem provides a performance improvement, but just over 2× the performance of the baseline memory subsystem.

5.4 Matrix-Matrix Multiply

As shown in Figure 8, the superoptimized memory subsystem for the matrix-matrix multiply benchmark (mm) provides about a 2× speedup over the baseline benchmark. For this benchmark, only the Distribute kernel uses external

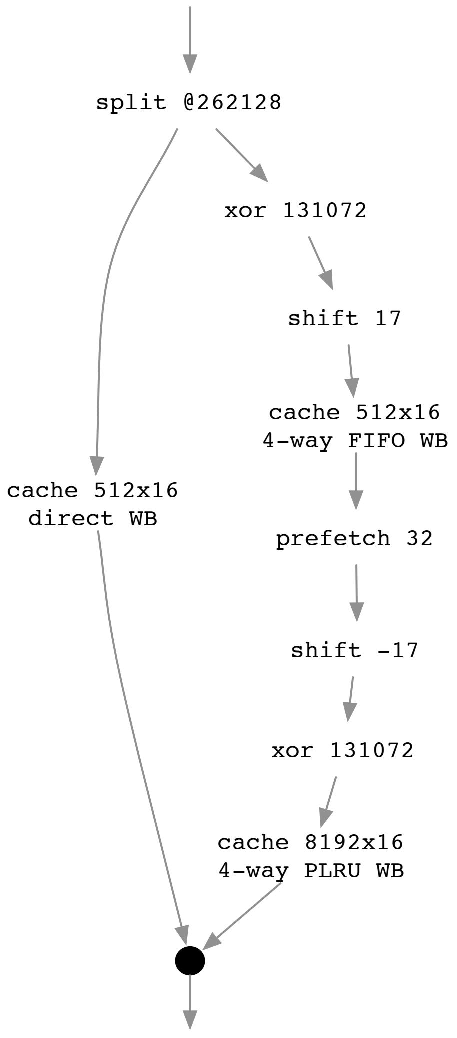

Figure 11: Subsystem for the `Distribute` kernel

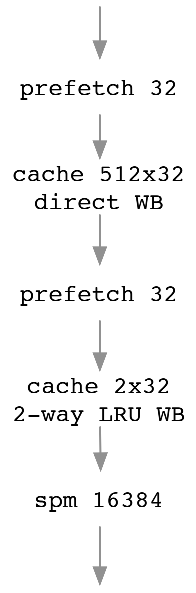

Figure 12: Subsystem for the `Buffer` kernel

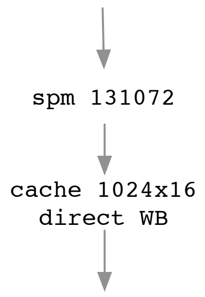

Figure 13: Subsystem for the `Streamer` kernel

memory. The superoptimized memory subsystem for the `Distribute` kernel is shown in Figure 11.

There are several interesting features of the memory subsystem shown in Figure 11. The first observation is the split. The split causes the memory accesses for the two source matrices to go to separate caches. The left side of the split handles the matrix that is accessed in column-major order whereas the right side handles the matrix that is accessed in row-major order. After the split, the first matrix is stored in a cache, whereas the second matrix is transposed from the memory subsystem's perspective before entering a cache.

All but four FIFOs are implemented as registers in the superoptimized memory subsystem for the mm benchmark. The two FIFOs between the `Distribute` kernel and the `Product` kernels are 256 entries and implemented in BRAM. The FIFOs between the `Product` kernels and `Combine` kernel are 128 entries deep and implemented in BRAM as well.

5.5 n-body

For the n-body benchmark, neither the simulated nor actual speedup are very large. This is because the n-body benchmark is compute-bound. However, we note that there is a performance gain even in this case. For this benchmark, all of the FIFOs are implemented as single-element registers. This allows all of the memory resources to be dedicated to the two kernel memory subsystems.

The superoptimized memory subsystem for the `Buffer` kernel is shown in Figure 12. This memory subsystem contains two prefetch components, two caches, and a scratchpad. The first prefetch requests the value 32 bytes after the current address, which causes the first cache to request the next line after the current access. Likewise, the second prefetch has the same effect on the second cache. Finally, the scratchpad stores the first elements rather than storing everything in main memory.

The memory subsystem for the `Streamer` kernel, shown in Figure 13, is a scratchpad followed by a cache. Unlike the previous memory subsystem, here the scratchpad comes first. This is likely due to the fact that placing the scratchpad after a cache, as is done for the memory subsystem for the `Buffer` kernel, incurs extra latency and poisons the cache. However, the prefetch components used for the `Buffer` kernel memory subsystem reduce this effect.

Given the way the benchmark works, it is not intuitive that the superoptimized memory subsystem for the `Buffer` kernel would be more complex than the subsystem for the

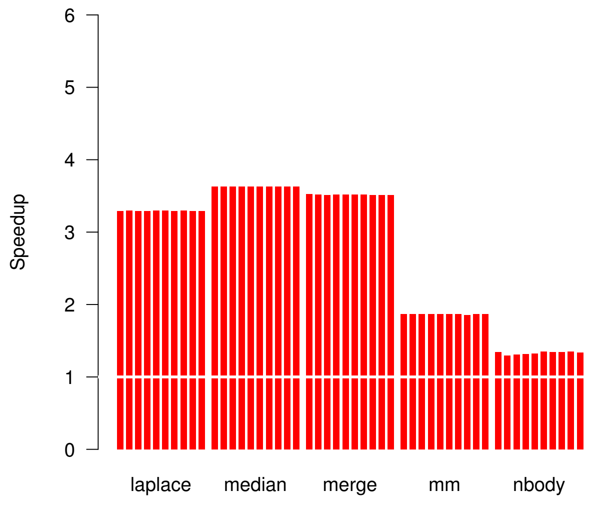

Figure 14: Input Specificity

`Streamer` kernel since the `Streamer` kernel streams the data past the `Force` kernel. However, because the `Force` kernel is computationally intensive, the memory delays that the `Streamer` kernel experiences contribute little to the overall run time. Instead, reducing the memory access times for the `Buffer` kernel provides a greater performance advantage than reducing the access times for the `Streamer` kernel.

5.6 Input Specificity

Although we are able to obtain a performance improvement for each of the benchmarks, we note that this improvement is not for the benchmark, but for a particular data set used with the benchmark. Because we are using only a single address trace for the optimization, it is possible that the memory subsystems could be over-fitted. Indeed, this appears to have happened for the `Hash` kernel for the median benchmark (Figure 9), which contains an address transformation to move parts of the hash table into a scratchpad.

To determine to what extent over-fitting affects the results, we re-ran each of the benchmarks with ten separate inputs. The results are shown in Figure 14. Here each bar shows the speedup of the superoptimized memory subsystem over the baseline memory subsystem for a particular data set. The left-most bar in each group shows the result from the original data set presented above. The nine remaining bars show the speedup for different data sets.

To change the input for the Laplace benchmark, we used a different random number seed. As shown in Figure 14, using a different random number seed has little effect on the speedup. For both the median and merge benchmarks, we used different data sets of the same size as the original. Finally, for the n-body benchmark, we used a different input size for each run (1,000 to 10,000 in increments of 1,000).

As Figure 14 shows, there is very little difference in the performance gain with different input data sets. This suggests that the superoptimized memory subsystems are not over-fitted. Nevertheless, it is conceivable that some superoptimized memory subsystems could be overly specific for a particular data set. In some cases, this could be desirable. For example, if an application used a hash table and the data stored in the hash table never changed. Generally, overfitting is something we would likely want to avoid.

5.7 Discussion

As the above results indicate, it is possible to superoptimize memory subsystems for streaming applications. The structure of some of the superoptimized memory subsystems are not surprising. For example, the memory subsystem for the Laplace benchmark is likely very similar to what one would select manually. On the other hand, some of the memory subsystems are logical, but would likely require manual experimentation to discover. For example, the memory subsystems for the median and the merge benchmarks are fairly standard, but require the tuning of many parameters. Finally, the superoptimizer is able to discover memory subsystems that are very unusual, such as those for the matrix-matrix multiply and n-body benchmarks.

The superoptimization process can take a long time, depending on the number of memory subsystems and the length of the memory address traces. The superoptimized memory subsystems presented here were generated by running the superoptimizer for between 10,000 and 200,000 simulation runs, depending on the benchmark. Applications with only a few memory subsystems, such as the Laplace benchmark, require far fewer simulation runs than those with many memory subsystems, such as the merge benchmark.

The run time of each simulation depends on the length of the address trace as well as the complexity of the memory subsystem. For the benchmarks presented here, the time for a single simulation is in the range of 5 to 15 minutes. To reduce the total run time for the superoptimization process, we made use of multiple processing cores and stored the results from each simulation in a database shared among the processing cores. The database allows the superoptimizer to revisit prior results without simulation.

As the superoptimization process runs, the memory subsystem it discovers never gets worse and is at all times usable. Thus, it is possible to terminate the process as soon as a satisfactory memory subsystem is discovered. The use of of multiple cores allowed us to obtain the results presented here in a week.

6. CONCLUSION AND FUTURE WORK

We have described a technique for creating superoptimized memory subsystems for streaming applications. We have shown that not only do these superoptimized memory subsystems perform well in simulation, but, by deploying the applications on an FPGA device, we have shown that these memory subsystems perform well in actual hardware. Through the use of ScalaPipe with our superoptimizer, we were able to create a design implemented on an FPGA device using a customized memory subsystem with minimal effort and without writing HDL.

In the future, we would like to explore the use of superoptimized memory subsystems with existing HDL designs. In addition, we would like extend this approach to optimize the memory subsystem for objectives other than performance, such as minimizing writes or energy consumption.

7. ACKNOWLEDGMENTS

This work was supported by Exegy, Inc., and VelociData, Inc. Washington University in St. Louis, R. Chamberlain, and R. Cytron receive income based on a license of technology by the university to Exegy, Inc., and VelociData, Inc.

8. REFERENCES

[1] M. Adler, K. E. Fleming, A. Parashar, M. Pellauer, and J. Emer. LEAP scratchpads: automatic memory and cache management for reconfigurable logic. In *Proc. of 19th Int'l Symp. on Field Programmable Gate Arrays*, pages 25–28, 2011.

[2] R. Balasubramonian, D. H. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. A dynamically tunable memory hierarchy. *IEEE Trans. on Computers*, 52(10):1243–1258, Oct. 2003.

[3] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In *Proc. of 10th Int'l Symp. on Hardware/Software Codesign*, pages 73–78, 2002.

[4] R. D. Chamberlain, M. A. Franklin, E. J. Tyson, J. H. Buckley, J. Buhler, G. Galloway, S. Gayen, M. Hall, E. B. Shands, and N. Singla. Auto-Pipe: Streaming applications on architecturally diverse systems. *Computer*, 43(3):42–49, Mar. 2010.

[5] R. D. Chamberlain and N. Ganesan. Sorting on architecturally diverse computer systems. In *Proc. of 3rd Int'l Workshop on High-Performance Reconfigurable Computing Technology and Applications*, Nov. 2009.

[6] E. S. Chung, J. C. Hoe, and K. Mai. CoRAM: an in-fabric memory architecture for FPGA-based computing. In *Proc. of 19th Int'l Symp. on Field Programmable Gate Arrays*, pages 97–106, 2011.

[7] J. Cong, M. Huang, and P. Zhang. Combining computation and communication optimizations in system synthesis for streaming applications. In *Proc. of 22nd Int'l Symp. on Field Programmable Gate Arrays*, pages 213–222. ACM, 2014.

[8] G. Dueck and T. Scheuer. Threshold accepting: a general purpose optimization algorithm appearing superior to simulated annealing. *Journal of Computational Physics*, 90(1):161–175, 1990.

[9] A. Ghosh and T. Givargis. Cache optimization for embedded processor cores: An analytical approach. *ACM Trans. on Design Automation of Electronic Systems*, 9(4):419–440, Oct. 2004.

[10] A. Gordon-Ross, F. Vahid, and N. Dutt. Automatic tuning of two-level caches to embedded applications. In *Proc. of the Conf. on Design, Automation and Test in Europe*, page 10208, 2004.

[11] A. Gordon-Ross, F. Vahid, and N. Dutt. Fast configurable-cache tuning with a unified second-level cache. In *Proc. of Int'l Symp. on Low Power Electronics and Design*, pages 323–326, 2005.

[12] T. C. Hu, A. B. Kahng, and C.-W. A. Tsao. Old bachelor acceptance: A new class of non-monotone threshold accepting methods. *ORSA Journal on Computing*, 7(4):417–425, 1995.

[13] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proc. of 17th Int'l Symp. on Computer Architecture*, pages 364–373, 1990.

[14] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simmulated annealing. *Science*, 220(4598):671–680, 1983.

[15] H. Massalin. Superoptimizer: a look at the smallest program. In *Proc. of 2nd Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 122–126, 1987.

[16] M. Matsumoto and T. Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. on Modeling and Computer Simulation*, 8(1):3–30, 1998.

[17] A. Naz. *Split Array and Scalar Data Caches: A Comprehensive Study of Data Cache Organization.* PhD thesis, Univ. of North Texas, 2007.

[18] P. R. Panda, N. D. Dutt, and A. Nicolau. Local memory exploration and optimization in embedded systems. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 18(1):3–13, 1999.

[19] P. R. Panda, N. D. Dutt, A. Nicolau, F. Catthoor, A. Vandecappelle, E. Brockmeyer, C. Kulkarni, and E. De Greef. Data memory organization and optimizations in application-specific systems. *IEEE Design & Test of Computers*, 18(3):56–68, 2001.

[20] E. Schkufza, R. Sharma, and A. Aiken. Stochastic superoptimization. In *Proc. of 18th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 305–316, 2013.

[21] J. H. Spring, J. Privat, R. Guerraoui, and J. Vitek. StreamFlex: high-throughput stream programming in Java. *ACM SIGPLAN Notices*, 42(10):211–228, 2007.

[22] K. T. Sundararajan, T. M. Jones, and N. P. Topham. Smart cache: A self adaptive cache architecture for energy efficiency. In *Proc. of Int'l Conf. on Embedded Computer Systems*, pages 41–50, 2011.

[23] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A language for streaming applications. In *Proc. of 11th Int'l Conf. on Compiler Construction*, pages 179–196, 2002.

[24] J. Vasiljevic and P. Chow. MPack: global memory optimization for stream applications in high-level synthesis. In *Proc. of Int'l Symp. on Field Programmable Gate Arrays*, pages 233–236, 2014.

[25] A. Veidenbaum, W. Tang, R. Gupta, A. Nicolau, and X. Ji. Adapting cache line size to application behavior. In *Proc. of 13th Int'l Conf. on Supercomputing*, pages 145–154, 1999.

[26] J. G. Wingbermuehle, R. D. Chamberlain, and R. K. Cytron. ScalaPipe: A streaming application generator. In *Proc. of 2012 Symp. on Application Accelerators in High-Performance Computing*, pages 244–254, 2012.

[27] J. G. Wingbermuehle, R. K. Cytron, and R. D. Chamberlain. Superoptimization of memory subsystems. In *Proc. of Conf. on Languages, Compilers, and Tools for Embedded Systems*, 2014.

[28] F. Winterstein, S. Bayliss, and G. Constantinides. Separation logic-assisted code transformations for efficient high-level synthesis. In *Proc of 22nd Int'l Symp. on Field Programmable Custom Computing Machines*, pages 1–8, 2014.

[29] H.-J. Yang, K. Fleming, M. Adler, and J. Emer. Optimizing under abstraction: Using prefetching to improve FPGA performance. In *Proc. of 23rd Int'l Conf. on Field Programmable Logic and Applications*, pages 1–8, 2013.