Spring 5-15-2015

Application-Specific Memory Subsystems

Joseph George Wingbermuehle
Washington University in St. Louis

WASHINGTON UNIVERSITY IN ST. LOUIS

Department of Computer Science & Engineering

Dissertation Examination Committee:
Roger D. Chamberlain, Chair
Kunal Agrawal
Ron K. Cytron
Viktor Gruev
Krishna Kavi
Hiro Mukai

Application-Specific Memory Subsystems

by

Joseph G. Wingbermuehle

A dissertation presented to the
Graduate School of Arts and Sciences
of Washington University in
partial fulfillment of the
requirements for the degree
of Doctor of Philosophy

May 2015

St. Louis, Missouri

Table of Contents

List of Figures

List of Tables

Acknowledgments

I would like to thank my adviser, Dr. Roger D. Chamberlain, for his excellent guidance. Our many discussions, his detailed feedback on manuscripts, and general advice have been invaluable to me. I am extremely grateful for his support.

I would like to thank my co-adviser Dr. Ron K. Cytron. His wealth of ideas and encouragement have proven indispensable in helping me complete my dissertation and I appreciate them greatly.

I am grateful for the financial support provided by NSF awards CNS-09095368 and CNS-0931693 as well as the financial support provided by Exegy Inc. and VelociData Inc. This support allowed me to focus exclusively on my research.

I would like to thank the members of my dissertation committee, whose valuable feedback helped define and focus my research.

I would like to thank my parents, George and Elaine Wingbermuehle, for their encouragement and support.

Finally, I would like to thank my partner, Ryan Richt, for his awesome ideas, unparalleled patience, and unwavering support throughout my time as a graduate student.

Joseph G. Wingbermuehle

Washington University in St. Louis

May 2015

ABSTRACT OF THE DISSERTATION

Application-Specific Memory Subsystems
by
Joseph G. Wingbermuehle
Doctor of Philosophy in Computer Science
Washington University in St. Louis, 2015
Professor Roger D. Chamberlain, Chair

The disparity in performance between processors and main memories has led computer architects to incorporate large cache hierarchies in modern computers. These cache hierarchies are designed to be general-purpose in that they strive to provide the best possible performance across a wide range of applications. However, such a memory subsystem does not necessarily provide the best possible performance for a particular application.

Although general-purpose memory subsystems are desirable when the work-load is unknown and the memory subsystem must remain fixed, when this is not the case a custom memory subsystem may be beneficial. For example, in an application-specific integrated circuit (ASIC) or a field-programmable gate array (FPGA) designed to run a particular application, a custom memory subsystem optimized for that application would be desirable. In addition, when there are tunable parameters in the memory subsystem, it may make sense to change these parameters depending on the application being run. Such a situation arises today with FPGAs and, to a lesser extent, GPUs, and it is plausible that general-purpose computers will begin to support greater flexibility in the memory subsystem in the future.

In this dissertation, we first show that it is possible to create application-specific memory subsystems that provide much better performance than a general-purpose memory subsystem. In addition, we show a way to discover such memory subsystems automatically using a *superoptimization* technique on memory address traces gathered from applications. This allows one to generate a custom memory subsystem with little effort.

We next show that our memory subsystem superoptimization technique can be used to optimize for objectives other than performance. As an example, we show that it is possible to reduce the number of writes to the main memory, which can be useful for main memories with limited write durability, such as flash or Phase-Change Memory (PCM).

Finally, we show how to superoptimize memory subsystems for streaming applications, which are a class of parallel applications. In particular, we show that, through the use of ScalaPipe, we can author and deploy streaming applications targeting FPGAs with superoptimized memory subsystems. ScalaPipe is a domain-specific language (DSL) embedded in the Scala programming language for generating streaming applications that can be implemented on CPUs and FPGAs. Using the ScalaPipe implementation, we are able to demonstrate actual performance improvements using the superoptimized memory subsystem with applications implemented in hardware.

Chapter 1: Introduction

As processors become faster and more numerous, memory access time is increasingly becoming the biggest bottleneck for many applications [80, 127]. To combat this performance gap between processing engines and main memory, modern computers employ large cache hierarchies. This situation has advanced to the point where 40% to 50% of the area [12] and up to 75% of the power budget [118] of a modern processor is dedicated to caching.

By exploiting both temporal and spatial localities in memory references, cache hierarchies are able to reduce the number of accesses to main memory, and, therefore, reduce memory access time. In this way, cache hierarchies are often able to greatly improve the performance of applications, explaining their prevalence [103]. However, in many cases, the application must be modified to expose locality to the cache hierarchy [10, 36, 67, 100]. In addition, the best cache parameters for one application are not necessarily ideal for all applications [70, 79]. Finally, certain classes of applications have little or no locality to exploit.

Although cache hierarchies are ubiquitous in general-purpose computers today, other types of memory components could also be considered. Indeed, modern processors often include other components, such as prefetchers [30, 53]. Also, scratchpads [5] are common in embedded systems. This leads us to the notion of a generalized memory subsystem. A generalized memory subsystem could contain caches, prefetchers, scratchpads, and possibly other components, with the goal of providing some form of improvement over direct access to main memory. Thus, here we define a *memory subsystem* as an on-chip memory that sits between a computation unit (such as a CPU, GPU, or FPGA) and off-chip main memory.

To provide a motivating example, consider matrix-matrix multiplication. Performing matrix-matrix multiplication is an important step in many applications. Unfortunately, matrix-matrix multiplication is computationally intensive for large matrices. Worse, a naive implementation typically has extremely poor cache performance. For these reasons, matrix-matrix multiplication has been a popular choice in benchmarks, such as LINPACK [33], and scientific libraries, such as BLAS [32], for many years.

Due to the need for fast matrix-matrix multiplication, the problem is well-studied [37, 43] and cache-efficient algorithms exist. However, these cache-efficient algorithms are more difficult to implement than the naive algorithm. Further, the techniques used to improve the access patterns of matrix-matrix multiplication do not generalize to all problems, leaving us to start over as soon as we are presented with a new problem.

If we were to implement matrix-matrix multiplication on an FPGA without considering how it worked, a cache would be a likely memory subsystem choice. Unfortunately, due to the access patterns of a naive matrix-matrix multiplication implementation, a cache would provide only a limited benefit. One way to improve the situation would be to modify the algorithm to make better use of the cache at our disposal or tune the cache parameters to better accommodate the algorithm. However, if we extend our search to other memory subsystem components, we might arrive at a more appropriate memory subsystem without needing to change the algorithm. In addition, if we were able to perform this search automatically, such a technique could require very little effort on the part of the designer and would be applicable to a wide range of problems.

Because of the potential improvement that a custom memory subsystem may provide in terms of performance, energy, or other metrics, we propose the use of a memory subsystem tailored to a particular application. Such custom memory subsystems are already in wide use today in applications deployed on Application-Specific Integrated Circuits (ASICs)

and Field-Programmable Gate Arrays (FPGAs) [5, 41, 94] as well as embedded systems in general [9]. Further, it is conceivable that general-purpose computer systems may one day be equipped with a more configurable memory subsystem if such reconfigurability provided enough of an advantage.

With a cache, one possible customization involves changing the cache parameters, such as the line size, associativity, or replacement policy. Selecting the optimal parameters for custom cache hierarchies is commonly done and remains an active area of research [39, 48, 56]. However, in our example application and in general, there is no reason to believe a traditional cache hierarchy would perform better than some other memory subsystem structure, such as a scratchpad.

Given our hypothetical matrix-matrix multiply application to be deployed on an FPGA, the person tasked with the design of the memory subsystem might select a small set of likely candidate designs and then perform some number of simulations to tune the designs and select the best. Unfortunately, this process is labor intensive for the designer and, worse, it is possible that the optimal design is not even considered. Ideally, this process could be automated in a way that provides a custom design beyond a fixed candidate memory structure. Therefore, our goal is to start with an empty memory subsystem, and add caches, scratchpads, address transformations, splits, and other components to the memory subsystem to arrive at an optimal memory subsystem, given the memory subsystem components at our disposal.

Using the techniques described in this work, we are able to design custom memory subsystems for applications, such as matrix-matrix multiply, that can out-perform generic memory subsystems such as cache hierarchies. For matrix-matrix multiply, one of the best-performing memory subsystems discovered by the work presented here contains not only a cache, but

also address transformations to "transpose" one of the matrices (described in detail in Chapter 4).

This research draws motivation from superoptimization, which was introduced with the goal of finding the smallest instruction sequence to implement a function [76]. Superoptimization differs from traditional program optimization in that superoptimization attempts to find the best sequence of instructions to implement a particular function at the expense of a potentially long search process rather than simply improving upon an existing sequence of instructions using a brief transformation process.

Traditionally, superoptimizers have used exhaustive search, however, as the search space gets larger, exhaustive search becomes prohibitively time-consuming. To address this issue, the notion of stochastic superoptimization [99] was introduced. Using stochastic superoptimization, one is able to discover larger instruction sequences, however, we lose the guarantee of finding the best instruction sequence in finite time. Fortunately, in practice stochastic superoptimization provides good results.

In this work we are concerned not with optimal instruction sequences, but instead with optimal memory subsystems. Therefore, although historically superoptimization has been defined as the search for the optimal code sequence to implement a function, here we generalize the definition as follows:

> *Superoptimization* is the search for a near-optimal design solution with little structural restriction at the expense of substantial search time.

Thus, as an example, with traditional superoptimization all combinations of instructions are considered rather than only those sequences that a particular compiler knows how to generate. For our purposes, we consider all memory subsystem components that the superoptimizer is capable of considering rather than a fixed memory structure.

Our initial investigation focuses on the discovery of application-specific memory subsystems providing the lowest possible execution time for single-threaded applications. To that end, using a memory address trace from the application, we use a stochastic superoptimization technique to discover a suitable memory subsystem. The discovered memory subsystems can be very unusual, but always provide at least as good of performance as a traditional cache and usually better.

We also show that it is possible to superoptimize a memory subsystem for objectives other than performance. In particular, we show that it is possible to reduce writes to main memory. Such an objective is important for certain types of memory technologies whose lifetime is limited by the number of writes, such as flash [11] and Phase-Change Memory (PCM) [126].

Because modern computer systems are becoming increasingly parallel, we next investigate the use of application-specific memory subsystems for parallel applications. In particular, we focus on streaming applications, which are a class of parallel applications that are particularly well-suited for implementation on ASICS, on FPGAs, and in heterogeneous hardware settings [19]. Streaming applications provide several additional challenges for memory subsystem superoptimization, including the communication between kernels and the enormous search space. Nevertheless, using heuristics and a queuing model, we are able to superoptimize the memory subsystems for streaming applications in a reasonable amount of time.

1.1 Research Questions

In this dissertation we attempt to answer the following research questions:

- Can application-specific memory subsystems provide a performance improvement over general-purpose memory subsystems?
- Is it possible to discover automatically application-specific memory subsystems?
- Can application-specific memory subsystems be beneficial for other main memory technologies, such as phase-change memory?
- Can application-specific memory subsystems be discovered for parallel applications?
- What can be done to reduce the time to find application-specific memory subsystems?

1.2 Contributions

To answer these research questions, this work makes the following contributions:

- ScalaPipe, which is a tool for generating streaming applications [120, 121].
- A tool to simulate quickly address traces using arbitrarily complex memory subsystems [122, 123].
- A method for the superoptimization of memory subsystems for single-threaded applications [122, 123].
- An evaluation of memory subsystems superoptimized to minimize writes to main memory as well as memory subsystems superoptimized for multiple objectives.
- A method for extending the superoptimization process to support the superoptimization of memory subsystems for streaming applications [124].
- An comparison of application-specific memory subsystems and general-purpose memory subsystems for applications implemented on an FPGA device [124].

- A queuing model to reduce the number of events that need to be simulated for the superoptimization of memory subsystems for streaming applications.

1.3 Outline

The remainder of this dissertation is organized as follows: Chapter 2 introduces background and related work. Chapter 3 describes the tools built to explore this area. Chapter 4 describes our superoptimization technique and how to apply it to simple single-threaded applications implemented in ASICs and FPGAs. Chapter 5 extends the superoptimization technique to a class of parallel applications and provides an evaluation of the technique for applications implemented on an FPGA device. Chapter 6 describes and evaluates a model to allow faster superoptimization of parallel applications. Finally, Chapter 7 provides conclusions and future work.

Chapter 2: Background and Related Work

In this chapter we provide a background for some of the concepts that we will use in later chapters. In particular, we provide an overview of both on-chip and off-chip memories. We then describe the various memory subsystem components that we consider for superoptimization. Finally, we present related work.

2.1 On-Chip Memory

In this work, it is useful to make a distinction between on-chip and off-chip memories. *On-chip* memory is memory that is present on the same die as the processing unit, where a processing unit might be a general-purpose processor, GPU, ASIC, or FPGA. As an example, on an FPGA on-chip memory is typically available in the form of block-RAM (BRAM). *Off-chip* memory, on the other hand, is memory that is physically separate from the processing unit, for example, the main memory in a general-purpose computer.

Because on-chip memory is co-located with the processing unit, it is typically much smaller than off-chip memory due to space limitations. However, on-chip memory is usually much faster than off-chip memory since there is no need to access a physically distant component. In addition, access to on-chip memory happens over separate data and address lines, which allows fast access and makes the interface to on-chip memory relatively simple compared

to that of off-chip memories, which will be described in the next section. Due to these advantages, memory subsystems, such as caches and scratchpads, are typically implemented in on-chip memory.

Static random-access memory (SRAM) is the most common memory technology used with on-chip memories [55]. SRAM is a volatile memory in that it requires a source of power to maintain its contents. Further, SRAM uses a relatively large area, typically using six transistors per bit, and it is power-hungry. Nevertheless, SRAM is popular because it is very fast and it uses the same fabrication process as a typical microprocessor.

Due to the the prevalence of SRAM, we assume that all on-chip memory is implemented as SRAM for our experiments. Further, we will use only on-chip memory for the implementation of memory subsystem components, such as caches and scratchpads. Off-chip memory will be used exclusively for the main backing store that stores the whole memory image, which we call *main memory*.

2.2 Off-Chip Memory

Off-chip memory is memory that is physically separate from the processing unit. Since off-chip memory is located on a separate physical device, potentially spanning multiple devices, the off-chip memory can be larger than is possible with on-chip memory. However, this separation limits performance due to wire delays, large multiplexers, and the fact that the physical pins of the device are used for multiple purposes to reduce pin count. Here we give only a high-level overview of the operation of a typical main memory implemented as off-chip memory. See [55] for a thorough treatment.

Figure 2.1 shows a typical main memory layout. Here, memory cells are arranged into *rows* (also known as *pages*) and *columns*. Each memory array is known as a *bank* and multiple

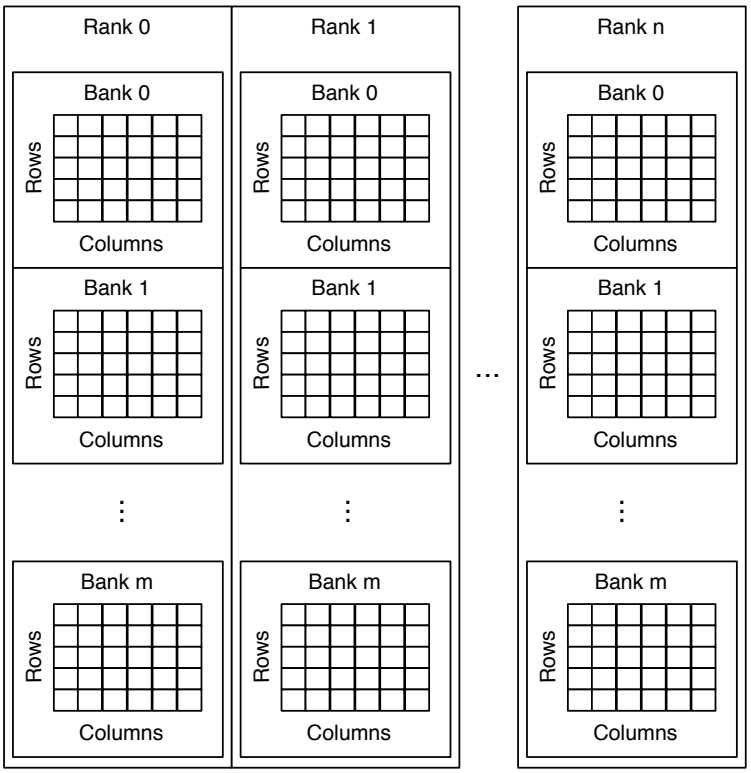


Figure 2.1: Main Memory Layout

banks are combined to form *ranks*. In addition, multiple *channels* can be provided to allow greater parallelism in the main memory.

Figure 2.2 shows an example of how a memory address might be divided up to access main memory. In this example, there are four ranks selected using bits 29 and 30. Within each rank there are four banks selected using bits 14 and 15. Finally, within each bank there is a memory array consisting of 8192 rows and 512 columns. Each column is 8 bytes, or 64 bits, making each page 32,768 bits.

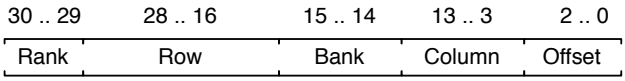


Figure 2.2: Main Memory Addressing

In the interest of keeping pin count down, the column and row addresses as well as the data are multiplexed over the same physical pins of the device. Thus, to access a word, first the

row address is sent to the appropriate bank, which loads the row into the *row buffer*. Next, the column address is sent to the memory, which selects the portion of the row buffer to read or write. Note that for most devices, it is necessary to *precharge* the bitlines before selecting a row. This is necessary to prevent the wrong value from being read if the memory cells have a weak influence on the bitlines.

In a memory arrangement such as shown in Figure 2.1, each rank operates in lockstep. The banks, however, each have their own row buffer, allowing multiple requests to be serviced in parallel. Further, once loaded into the row buffer, it is possible to access multiple columns of the selected row without selecting a new row or precharging the bitlines. Keeping rows open in this fashion is known as *open-page* mode. Open-page mode has benefits when multiple accesses hit in the same row. On the other hand, *closed-page* mode is when the row is not held open. In particular, with closed-page mode the bitlines are precharged immediately after an access to prepare for the next access, which allows the device to access the next row more quickly. Thus closed page mode is faster if multiple hits in the same row are unlikely.

Another performance improvement that is common is reading bursts of data from the memory. Once the row buffer is loaded for a read or write, multiple words can be accessed at a time. This is known as a *burst*. For example, if the size of a word is 16 bits and the burst size is 4, every access will consist of 64 bits.

Most modern devices are *synchronous* rather than *asynchronous*. This means that all operations on the device are managed by a fixed clock. Further, *double-data rate* (DDR) devices are prevalent today. A double-data rate device transfers data on both the rising and falling clock edges, allowing it to transfer two times more data than the clock would otherwise indicate.

Parameter	Description
Frequency	DRAM I/O frequency
CAS	Cycles to select a column (Column-address strobe)
RCD	Cycles from row select to access (RAS-CAS delay)
RP	Cycles required for precharge (RAS precharge)
Page size	Size of a page in bytes
Page count	Number of pages per bank
Width	Channel width in bytes
Burst size	Number of columns per access
Page mode	Open or closed page mode
DDR	Double data rate

Table 2.1: Main Memory Parameters

Modeling of an off-chip memory device requires consideration of several important timing parameters. These timing parameters are summarized in Table 2.1. Note that there are more timing parameters to consider, especially for modern, high-speed devices. Such parameters are essential for correct operation when designing a memory controller, but for our purposes we consider this simplified model.

In Table 2.1, the frequency is the I/O frequency of the device (assuming synchronous operation). The *CAS* (column-address strobe) latency is the number of cycles required between selecting a column and reading the data. The *RCD* (RAS-CAS delay) latency is the number of cycles required between selecting a row and selecting a column (note that *RAS* stands for row-address strobe). The *RP* (RAS precharge) latency is the number of cycles required to precharge a row. Finally, the page size is the size of each row and the page count is the number of pages per bank.

Next we provide an overview of several competing technologies used to implement main memory cells.

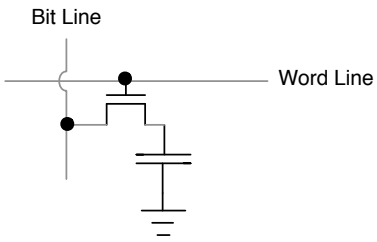


Figure 2.3: DRAM Cell

2.2.1 DRAM

Due to its low cost and small size, dynamic read-only memory (DRAM) is the most common main memory technology in use today [98]. Because of its prevalence, for most of the experiments presented here we assume that the main memory is a DRAM device, though we note that our superoptimization technique is generic and could be used with another memory model.

DRAM works by storing charge in a capacitor. The typical DRAM cell consists of a single transistor and a capacitor, shown in Figure 2.3. To access a word, first the bitlines are precharged to an intermediate value. This is done because the charge on the capacitors is weak, thus precharging to an intermediate value allows the device to detect if the charge on the capacitor is pulling the bitline voltage up or down. Next, the row is selected, which activates all the transistors on the row, thereby connecting the capacitors in the row to the bitlines.

Selecting a row loads all the bits of the row into the sense amplifiers, which are integrated with the row buffer (note that there is a sense amplifier circuit for each bit in a row). The sense amplifiers detect if each bit is a one or a zero based on the effect of the capacitor on the precharged bitline. In addition to reading the value from the bitlines, the sense amplifier recharges the capacitor. Finally, once the requested row is loaded into the sense amplifiers, the column is selected.

Another aspect of a DRAM device is refresh. A DRAM device must be refreshed periodically (typically once every 64 milliseconds) to preserve the charge stored in the capacitors. The process of refreshing is accomplished by accessing every row of the device, which causes the capacitor to be recharged due to the design of the sense amplifier. Refresh can require a significant amount of time that could be used for memory accesses, especially with larger DRAM devices [109]. However, since we do not have control over refresh, we ignore it in our performance model.

DRAM has long been the technology of choice for main memories, but there is ample room for improvement. Three problems with DRAM include scaling, energy, and volatility. Because DRAM stores charge in a capacitor, there is a limit to scaling due to the ratio of the capacitance between the cell and the bitline [74]. In addition, the cell capacitor requires a continuous refresh, making DRAM devices volatile and inefficient from an energy perspective.

2.2.2 Phase-Change Memory

Phase-change memory (PCM) is a newer memory technology that is often cited as a potential replacement for DRAM [69, 93, 128, 135, 136]. Rather than storing charge in a capacitor as is the case with DRAM, PCM uses chalcogenide glass, whose state can be altered by heating it and then cooling it according to different temperature schedules [126]. When the chalcogenide glass is heated to a high temperature and cooled rapidly it remains in an amorphous state, which has a high resistance. When heated just above the crystallization threshold and held at that temperature for somewhat longer time, however, the chalcogenide glass stays in a crystalline state, which has a lower resistance. States between the two extremes are also possible, making the way for multi-level cells (MLCs), which allow the storage of more than one bit per cell.

Some benefits of PCM over DRAM include the fact that PCM is non-volatile and that PCM scales better than DRAM, since PCM does not use a capacitor to store charge and it can store multiple bits per cell. Because PCM is non-volatile, it does not require a refresh, which is costly in terms of energy and performance, especially as main memories get larger. Unfortunately, both reads and writes to PCM require more energy and time than the equivalent access to DRAM. Further, PCM has limited write endurance of around 10^8 write cycles per cell, which limits the lifetime of the part.

2.2.3 Flash

Flash memory [11, 49] is another non-volatile technology with some similar properties to PCM. Flash memory stores data by storing charge in a floating-gate transistor, whose floating gate is capable of storing charges for years. To introduce a charge to the floating gate, a high voltage is passed through the transistor, which causes some electrons to get trapped in the floating gate. To clear the charge, a high-voltage in the opposite direction is applied. As with PCM, multi-level cells are possible with flash memory.

Unfortunately, like PCM, flash memory has a limited write-endurance, however, whereas PCM has a write endurance of around 10^8 write cycles, flash has an endurance of only around 10^5 [69]. In addition, writing to flash is slow and requires a lot of energy.

Flash memory comes in two varieties: NAND and NOR. With NOR flash devices, each cell is connected to ground. This allows each bit to be written individually. However, the extra ground connection limits the density of NOR flash devices. Thus, NAND flash was developed, where several cells are connected in series. This makes for improved density, but requires that all transistors in the series be erased together. Despite this pitfall, NAND devices are more popular today [21].

Due to the high energy and time required to erase flash as well as its limited write endurance, flash is generally used as disk replacement rather than a DRAM replacement. As a disk replacement, NAND flash has been extremely successful.

2.2.4 STT-RAM

Finally, we consider spin-transfer torque RAM (STT-RAM), which is an emerging memory technology. Like PCM, STT-RAM is often cited as being a replacement for DRAM [64, 128]. Unlike other technologies, STT-RAM stores data in a magnetic field. Although STT-RAM lacks the write endurance issues of PCM and flash, writes to STT-RAM are slow and consume significant energy.

2.3 Memory Components

Here we give a brief overview of some of the memory subsystem components that we will consider in later chapters. There is, of course, an endless supply of memory subsystem components that one could consider. In addition, there are more generic forms of the components that we describe here, which would provide more parameters to tune. Here we describe those components and parameters that we will consider in the superoptimization process. One could extend the superoptimizer to support a wider array of components and parameters.

2.3.1 Caches

Caches are small memories that are used to store recently used data from main memory [103]. To accomplish this, caches are typically organized into *lines*, such that each line can store

some contiguous chunk of of words from main memory. A typical line size is 64 bytes. Since the line could be associated with multiple addresses in main memory, the most significant bits of the address must also be stored with each line.

Associativity

A cache in which each line can be associated with any address in main memory is called a *fully-associative* cache. Unfortunately, looking up a data element in such a cache can be too time consuming and require a significant amount of hardware to implement. Therefore, caches are often *set-associative*. This means that some fixed number of lines are consulted for each memory reference rather than all of the lines of the cache. For example, in a cache that is 4-way set associative, each memory address can be stored in one of four possible lines in the cache. Finally, a *direct-mapped* cache is a cache in which each memory address can only be stored in one line in the cache. There are trade-offs between the associativity of the cache and the hit rate for the application [110]. Caches with lower associativity use fewer resources and are often faster than highly-associative caches.

Replacement Policies

For caches that are either fully-associative or set-associative, we have to decide which line to evict when a new line is brought into the cache. This decision is known as the *replacement policy* of the cache. Here we consider the most popular cache replacement policies, but we note that there are many others [22, 57, 61, 92, 134]. The best policy depends on both the resource usage required to implement the policy and the behavior of the application [2].

Perhaps the most common replacement policy is the least-recently-used (LRU) policy. With an LRU cache, the line that has been accessed least recently is selected for replacement first.

This type of policy is intuitive since items that are often used will stay in the cache. Unfortunately, it requires $\lceil \lg n \rceil$ bits of storage per line to implement where n is the associativity of the cache.

Similar to the LRU policy is the most-recently-used (MRU) policy. With an MRU cache, the line that has been most recently used is evicted first. This policy seems like it would rarely be beneficial, and indeed, that is often the case. Nevertheless, it is possible to construct a memory access sequence that would benefit from such a policy. As with the LRU policy, $\lceil \lg n \rceil$ bits of storage per line are required.

Another common policy is the first-in first-out (FIFO) cache policy (also known as the round-robin policy), which has been used in commercial products such as the Intel XScale [52] and ARM ARM11 processors [3]. With a FIFO policy, the oldest line in the set is replaced. Unlike the LRU policy, accesses to the line after it has been brought into the cache do not affect the replacement decision. Since a FIFO cache requires only a counter per set to determine the next line to evict, the FIFO policy requires only $\lceil \lg n \rceil$ bits of storage per set where n is the associativity of the cache.

Finally, we consider a pseudo-LRU (PLRU) policy, which attempts to approximate the true LRU policy, but with simpler hardware. There are several ways of implementing a PLRU policy. Here we consider the method where a single bit of storage per line is used. This bit is set every time a cache line is accessed. If the bit for all the lines in a set are set, the bits are reset. Upon replacement, the first line with a unset bit is selected. Although this technique does not implement true LRU, it can allow a higher associativity than could be used with a true LRU policy due to resource constraints and it can be faster due to simpler hardware.

Write Policies

In general, reads from a cache are either serviced by the cache directly in the case of a hit, or cause a line to be replaced in the case of a miss. However, there are more options available for writes to a cache.

The first decision regarding the write policy is whether the cache should be write-through or write-back. On a *write-through* cache, all writes go directly to the next memory in the hierarchy whereas with a *write-back* cache, writes are cached and only cache lines evicted from the cache get written to the next memory. Both write-through and write-back caches have advantages. A write-back cache can reduce the amount of write traffic to the next memory. On the other hand, a write-through cache can avoid cache pollution and, on multiprocessor systems, a write-through cache makes coherency simpler.

Another write policy decision is how lines are allocated on writes if the write is a miss. If on a cache miss a line is allocated, we say that the cache is *write-allocate*. Otherwise, if no line is allocated, that is, the write goes directly to the next memory without allocating a line in the cache, we say that the cache is *write-over*. Typically, write-through caches use write-over and write-back caches use write-allocate.

2.3.2 Scratchpads

A *scratchpad* is a small, fast memory that handles memory accesses to a fixed portion of the address space [5]. Scratchpads find most use in embedded devices since they are easy to implement and have deterministic access times (unlike caches). The use of scratchpads is somewhat limited, however, because their use typically requires either manual programmer intervention or a custom compiler [114].

2.3.3 Prefetchers

Prefetching provides a method to "hide" memory latency by requesting an item from memory before it is needed by the computation. There are many prefetching mechanisms that have been proposed for both hardware and software [115]. A simple hardware method that we consider here is prefetching a word n bytes away from the current word after each read. Such a prefetcher would likely cause too much cache pollution to be of use in a general-purpose setting, but could be useful in an application-specific setting in part of the memory subsystem.

2.3.4 Splits

A memory subsystem can be *split* such that addresses below a certain threshold go to a different set of memory subsystem components than addresses above the threshold [83]. For example, it may be desirable in an application to have the stack stored in a cache separate from the heap.

2.3.5 Address Transformations

Transforming the address is another technique that can be used in the interest of improving memory performance. Some transformations that could be used include adding a constant to the address, flipping one or more bits of the address, and rotating the address bits. Although it may seem that such a transformation would be unproductive, when combined with other components, such as scratchpads, address transformations could be potentially very useful. Note that it is often necessary to reverse a transformation to maintain correctness (when used within a split memory, for example).

2.4 Related Work

There is much related work spanning several broad categories. Here we evaluate the related work in each category.

2.4.1 Superoptimization

Superoptimization was originally introduced in [76]. In that work, exhaustive search was used to find the smallest sequence of instructions to implement a function. This is in contrast with traditional code optimization where pre-defined transformations are used in an attempt to improve performance. Note that traditional code optimization is not truly optimization in the classical sense, but instead simply code improvement. Superoptimization, on the other hand, does produce an optimal result when applied in this manner.

Because of its long run time, superoptimization is typically not applied to complete programs, but, rather, it is applied to a few critical functions. One of the examples in [76] is the `signum` function, which is defined as follows:

$$signum(x) = \begin{cases} 1 & \text{if } x > 0 \\ -1 & \text{if } x < 0 \\ 0 & \text{otherwise} \end{cases}$$

When run through the superoptimizer in [76], it turns out this function can be implemented for the Motorola 68020 microprocessor [73], using only four instructions (shown below), whereas a naive implementation would take eight and a clever implementation would take six.

```
; x in d0
add.l   d0, d0      ; Add d0 to itself
subx.l  d1, d1      ; Subtract (d1 + carry) from d0
negx.l  d0          ; Put (0 - d0 - carry) into d0
addx.l  d1, d1      ; Add (d1 + carry) to d1
; signum(x) in d1
```

Unlike prior implementations by compiler backends or humans, this implementation is very unusual and much more efficient. Although it is conceivable that a human would devise such an implementation, it would likely require significant effort with no guarantee of success. Using a superoptimizer, on the other hand, requires minimal human effort.

Functions such as `signum` appear in many programs, and, therefore, it is advantageous for a compiler to have a fast implementation available. Since its introduction, superoptimization has been successfully used in compilers such as GCC [44], peephole optimizers [6], and binary translators [7]. However, this body of work is the first to expand the scope of superoptimization beyond the optimization of instruction sequences.

Because of the enormous search space, there have been a few attempts to reduce the number of points in the search space. Denali [59] uses a theorem prover to avoid testing incorrect instruction sequences and TOAST [13] uses answer set programming.

Another technique that has been used with superoptimizers is stochastic search [99]. This allows the superoptimizer to explore much larger search spaces than would be possible with exhaustive search. One disadvantage of such a technique is that the guarantee of optimality is lost unless the stochastic search is allowed to run for a very long time. Nevertheless, here we use a stochastic search technique to make the search for good memory subsystems tractable.

2.4.2 Design Space Exploration

Design space exploration for hardware and software systems is an active and wide area of research. The goal is of design space exploration is to find the optimal or near-optimal parameters for a particular system.

For designs with a large number of parameters, exploring the design space via exhaustive search can be intractable. Thus there exist several techniques to improve upon exhaustive search. In [54], the design space is sampled to build a model of the interactions between parameters. Regression modeling is used in [68] to predict the performance and power of various configurations. In [86], a method is presented for decoupling certain design parameters to reduce the search space.

Design space exploration has been applied to many fields, such as system-on-chip (SoC) communication architectures [65], integrated circuit design [129], FPGA designs [104], and many others [82, 106, 132]. Although a single objective, such as performance or energy, is often used, design space exploration for multiple objectives is also common [71, 87, 88].

For streaming applications targeting FPGAs, the optimization of both the computation and communication between kernels has been considered [28]. Similarly, in [125], 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, here we treat the computation as fixed, but consider a wider search space for memory subsystems.

Of particular interest to us is design space exploration applied to memory subsystems. Design space exploration has been used extensively to find optimal cache parameters [39, 48, 56]. This line of work has been extended to consider a cache and scratchpad together [23]. However, the ability to change completely the memory subsystem for a specific application and main memory subsystem distinguishes this work from previous work.

2.4.3 Software Techniques for Improving Memory Behavior

Here we are focused primarily on hardware techniques, however, there also exist software techniques for improving memory behavior. Such techniques include the use of profiling to guide the placement of variables in the virtual address space to decrease cache conflicts and improve locality [14] and compiler optimizations to improve data locality across loop iterations [15]. Other software techniques include reorganization and cache-conscious memory allocation [25] as well as the splitting and reordering of data structures [24].

At a higher level, there are approaches to application design that focus on improving cache performance. In particular, access ordering [81] and cache-aware algorithms [100] attempt to take advantage of a particular cache structure. Likewise, the performance of cache-oblivious algorithms [36] is asymptotically optimal on an ideal cache hierarchy.

Although these software methods are often successful at improving the memory behavior of an application with respect to a particular memory, in this work we treat the application as fixed. Thus, these software techniques can be considered complementary to this work.

2.4.4 Tuning Cache Parameters

Tuning cache parameters is an active research topic that is related because we, too, are tuning cache parameters. There are two broad types of work in this area: static methods and dynamic methods. With static parameter tuning, the properties of a fixed cache are selected before deploying the hardware. On the other hand, dynamic methods allow certain parameters of a cache to change at run time. Each technique has advantages and it is possible to mix them.

Dynamic methods include changing the size and associativity of a cache hierarchy dynamically [4, 111] as well as the ability to disable various levels of a multi-level cache in the interest of reducing latency and reducing power consumption [20]. Adjusting the size of cache lines dynamically to lower the cache miss rate has also been considered [117].

There are also many approaches to tuning cache parameters statically. A method for selecting cache parameters analytically has been described for single-level caches [40, 56]. In addition, heuristic methods for selecting the parameters of a two-level cache have been presented [41, 42].

Although we are mostly concerned with static parameter selection, dynamic methods could be incorporated into our superoptimizer to allow it to select such a cache. As far as the static methods are concerned, we note that, although we are considering a larger search space, it may be possible to incorporate such a technique into the optimizer to allow it to search the space of possible cache parameters more efficiently. Thus, our work is complementary to both types of cache parameter tuning.

2.4.5 Non-traditional Memory Subsystems

Many non-traditional memory subsystems have been proposed. These structures are often intended to be general-purpose in nature, but to take advantage of some aspect of application behavior that is common across many applications. However, there are also many non-traditional memory subsystems designed for particular applications, usually with much effort. Such designs are a common practice for applications deployed on FPGAs and ASICs [26, 35, 101]. Due to strict resource constraints, embedded systems in general commonly employ specialized memory subsystems [9].

One notable example of a general-purpose non-traditional memory subsystem is the victim cache [60], which is a small, fully-associative cache structure used to store recently evicted items from a larger cache with low associativity. Victim caches work on the assumption that some cache sets can benefit from a higher associativity than others. The use of such caches in embedded applications has been explored [133].

Another general-purpose memory subsystem is the annex cache [58]. The annex cache is similar to the victim cache, but instead of storing recently evicted items for a larger cache, the annex cache stores items that have yet to be moved into the main cache.

Although performance is perhaps the most common objective, non-traditional memory subsystems optimized for other objectives have also been considered. For example, the filter cache [62] was introduced to reduce energy consumption with a modest performance penalty. A filter cache provides a very small first-level cache in front of a second-level cache with a similar structure to a traditional first-level cache. Micro-caches [8] are similar to filter caches, but designed to provide performance/area efficiency in chip multiprocessors instead of energy efficiency on a single core. Note that the term microcache has been previously used in [78], which describes a method for reducing cache size and power consumption by allowing the compiler to allocate regions of the cache to specific objects.

The combination of multiple memory subsystem components has also been considered to various degrees. For example, the combination of a scratchpad and cache has been considered [89, 94]. Further, the combination of multiple caching techniques including split caches has been considered [83].

Unlike our work, these works present a particular memory subsystem. Our work, on the other hand, attempts to discover memory subsystems with arbitrary structure. Therefore, it is possible that our superoptimizer would discover similar structures if provided the necessary memory subsystem components.

2.4.6 Memory Interfaces

Finally, we consider related work in making off-chip memory easier to use. Implementing a memory intensive application in hardware using either an FPGA or ASIC can be a difficult task. This is due to the fact that off-chip memory bandwidth is limited and on-chip memory resources are scarce. Thus, designing a good memory subsystem requires one to efficiently allocate the on-chip memory and share the off-chip memory between various compute elements. As an additional complication, the interface to off-chip memory is platform-specific.

LEAP scratchpads [1] attempt to alleviate some of the issues with sharing memory resources among kernels by providing a portable memory abstraction. This memory abstraction may contain caching and can be backed by a larger main memory. A related approach is provided by CoRAM [27], which is similar to LEAP scratchpads, but lower-level. CoRAM 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. In the interest of improving the performance of such abstractions, prefetching [131] has been considered.

Both LEAP scratchpads and CoRAM are are similar to our work in that both provide an abstract interface to a potentially large memory. However, providing an interface is not our primarily goal. Here, we are more interested in discovering the memory subsystem to use between the interface and the off-chip memory.

A related technology is MPack [116], which 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.

Chapter 3: Tools

Here we discuss the tools that we developed for memory superoptimization. This includes tools for gathering address traces, simulating memory subsystems, superoptimizing memory subsystems, and deploying applications with custom memory subsystems. With the combined tool set described here (ScalaPipe, the memory simulator, the memory superoptimizer, and the memory generator), it is possible to take a design from a high-level language to an FPGA implementation with a custom memory subsystem without the need to write HDL.

3.1 ScalaPipe

Here we provide a brief overview of ScalaPipe [120, 121], which is a streaming application generator. For a more complete description see Appendix A.

Stream processing is a parallel programming paradigm in which processing kernels communicate over fixed communicate channels. The streaming paradigm is used in systems such as StreamIt [112] and many others [17, 29, 45, 46, 105]. Within the streaming paradigm, conceptually, each kernel has its own independent memory address space. Communication between kernels is performed via explicit communication channels implemented as FIFO buffers. Our interest in the streaming paradigm stems from our desire to superoptimize memory subsystems for parallel applications, as explained in Chapter 5.

```
val Adder = new Kernel {

    val x0 = input(UNSIGNED32)
    val x1 = input(UNSIGNED32)
    val y = output(UNSIGNED32)

    y = x0 + x1

}
```

Figure 3.1: Simple ScalaPipe Kernel

ScalaPipe provides a pair of domain-specific languages (DSLs) embedded in the Scala programming language [85]. By using ScalaPipe, one is able to author streaming applications that can then be deployed to a combination of CPUs and FPGAs. Using ScalaPipe to implement some of our benchmarks allows us not only to implement quickly an application that will run on an FPGA without the need to write in a hardware description language (HDL), but it also allows us to automatically extract an address trace, which we can use for superoptimization. Further, ScalaPipe allows us to deploy automatically the applications along with their superoptimized memory subsystems on an FPGA device.

To support the streaming paradigm, ScalaPipe allows one to author kernels in a *kernel* DSL and then describe the communication channels between kernels in the *application* DSL.

3.1.1 Kernel DSL

Here we describe ScalaPipe's kernel DSL. A simple kernel to add pairs of 32-bit unsigned integers is shown in Figure 3.1. This kernel has two inputs, `x0` and `x1`, and one output, `y`. Each time an input is referenced, a value is read off of the input channel associated with that input. Each time an output is referenced, a value is written to the output stream. Conceptually, ScalaPipe kernels run in an infinite loop processing data until all input is exhausted.

```
class GenericSplit(t: Type, n: Int) extends Kernel {
    val x = input(t)
    for (i <- Range(0, n)) {
        val y = output(t)
        y = x
    }
}
```

Figure 3.2: Generic Split Kernel

```
val app = new Application {
    val rng1 = Random()
    val rng2 = Random()
    val result = DivideBy2(Add(rng1, rng2))
    Print(result)
}
```

Figure 3.3: Averaging Application

Because a ScalaPipe kernel is implemented in the Scala programming language, it is possible to write generic kernels. For example, a kernel to divide an input stream of type `t` among `n` output streams is shown in Figure 3.2. Specific instances of this kernel can be created using `new`, just as one would create objects in Scala. Those familiar with Scala will note that we use `Range` explicitly in the `GenericSplit` kernel to force the loop to be unrolled before the code is generated. Thus, `n` outputs are created and the input is sent to each output in a round-robin fashion.

3.1.2 Application DSL

To connect kernels together to form a streaming application, ScalaPipe provides an *application* DSL. Using the application DSL, kernels are connected together much like function application, where the arguments to the kernel are the inputs and the result of the function application contains the outputs. For example, a simple application to average two streams of random numbers is shown in Figure 3.3.

By default, ScalaPipe will map all kernels to general-purpose processing cores and generate C code for their implementation. To map kernels to another resource, we can insert `map` statements. For example, to map everything but the `Print` kernel of our example application to a an FPGA device, we would use the following `map` statement:

```
map(ANY_KERNEL -> Print, FPGA2CPU())
```

This statement states that any edge entering a `Print` kernel will move from an FPGA resource to a CPU resource.

In addition to `map` statements, ScalaPipe supports various parameters that affect the way it generates code. Of particular interest to us is the *trace* parameter:

```
param('trace)
```

This parameter causes ScalaPipe to generate a memory address trace for each kernel when executing an application on a CPU resource. This address trace will be of use to us for the superoptimization process.

3.2 Memory Simulator

To evaluate custom memory subsystems, we developed a memory subsystem simulator. Unlike extant simulators, our simulator is capable of simulating arbitrarily complex memory subsystems and parallel address traces from streaming applications. The simulator is capable of evaluating multiple aspects of the memory system, including performance, writes to main memory, and the energy consumption of the main memory. It is also able to output compressed queue traces, which will be described in more detail in Chapter 6. As will be shown in Chapter 4, in addition to its ability to simulate complex memory subsystems, the ability for the simulator to run an address trace quickly is essential.

To support complex memory subsystems, the simulator takes a machine and memory subsystem description, encoded as S-expressions [72], as input. This encoding allows arbitrarily complex memory subsystems to be expressed. For example, Figure 3.4 shows a memory system for a streaming application with two kernels.

As shown in Figure 3.4, there are three main sections in the application description. The first section, `machine`, describes the platform on which the application will be run. In the example, this platform is a Xilinx Spartan-6 FPGA device running at 100 MHz. The second section of the application description is the `memory` section. This section describes the main memory (in this case, a DRAM device) as well as the memory subsystems for each kernel and communication channel. Finally, the `benchmarks` section points the simulator to the address traces for each kernel.

Our simulator is capable of simulating the memory subsystem components shown in Table 3.1. When targeting an FPGA device, the latency shown in Table 3.1 is used, which matches our VHDL implementation of the memory components. The CACTI tool [113] is used to determine latencies for ASIC targets. For the main memory, the simulator assumes that there is a priority arbiter in front of a main memory with a single read/write port.

For caches, the simulator supports four replacement policies. The supported policies include least-recently used (LRU), most-recently used (MRU), first-in first-out (FIFO), and pseudo-least-recently used (PLRU). The PLRU policy approximates the LRU policy by using a single *age* bit per cache way rather than $\lg n$ age bits, where n is the associativity of the cache. With the PLRU policy, the first way where the age bit is not set is selected for replacement. Upon access, the age bit for the accessed way is set and when all age bits are set for a set, all but the accessed age bit are cleared.

The `offset`, `rotate`, and `xor` components in Table 3.1 are address transformations. The `offset` component adds the specified value to the address. The `rotate` component rotates

```
(machine
    (target fpga)
    (part xc6slx45)
    (max_luts 12000)
    (max_regs 36000)
    (max_cost 92)
    (frequency 100000000)
    (addr_bits 30))
(memory
    (main (memory
        (dram
            (frequency 100000000)
            (cas_cycles 3)(rcd_cycles 3)(rp_cycles 3)
            (page_size 1024)(page_count 8192)
            (width 2)(burst_size 4)
            (open_page false)
            (ddr true))))
    (subsystem (id 1)(depth 65536)
        (memory (spm (size 8192)(memory (main)))))
    (subsystem (id 2)(depth 131072)
        (memory
            (split (offset 16384)
                (bank0
                    (cache
                        (line_count 1024)(line_size 8)
                        (associativity 1)
                        (write_back true)
                        (access_time 3)(cycles_time 3)
                        (memory (join))))
                (bank1
                    (cache
                        (line_count 1024)(line_size 4)
                        (associativity 2)(policy plru)
                        (write_back true)
                        (access_time 3)(cycles_time 3)
                        (memory (join))))
                (memory (main))))))
    (fifo (id 1)(depth 16)(word_size 4))
(benchmarks
    (trace (id 1)(name Kernel1))
    (trace (id22)(name Kernel2))
)
```

Figure 3.4: Example Memory Description

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 policy Write policy	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 3.1: Memory Subsystem Components

the bits of the address that select the word left by the specified amount (the bits that select the byte within the word remain unchanged). Note that for a 32-bit address with a 4-byte word, $32 - \lg 4 = 30$ bits are used to select the word. Finally, the `xor` component inverts the selected bits of the address.

Other supported components include `prefetch` and `split`. The `prefetch` component performs an additional memory access after every memory read to do the prefetch. This additional access reads the word with the specified distance from the original word that was accessed. Finally, the `split` component divides memory accesses between two memory subsystems based on address: accesses with addresses above a threshold go to a separate memory subsystem from addresses below the threshold. Accesses that are not resolved within the split are sent to the next memory subsystem or main memory.

3.3 Memory Superoptimizer

Because we are interested in the superoptimization of memory subsystems, we developed a superoptimizer to generate valid memory subsystems for simulation. This superoptimizer then uses the results of each simulation to generate another memory subsystem to check.

To ensure the validity of each memory subsystem, the superoptimizer is fed a list of constraints. The superoptimizer checks the constraints using an FPGA synthesis tool (such as the Xilinx Synthesis Tool, XST [130]) when targeting an FPGA device or CACTI [113] when targeting an ASIC. Because the superoptimization process can take a very long time, intermediate results are stored in a PostgreSQL [91] database.

The memory superoptimizer is described in detail in Chapter 4. Enhancements to the superoptimizer to allow the superoptimization of streaming applications are described in Chapter 5 and Chapter 6.

3.4 Memory Generator

Once we have a superoptimized memory subsystem, we need some way to deploy it for evaluation. Although it would be possible to manually create the superoptimized memory subsystems from the high-level description emitted from the superoptimizer, doing so would be tedious and error-prone. Therefore, we implemented a tool to generate the memory system automatically.

Our memory generator takes an S-expression [72] description of the memory system as described in Section 3.2 for input. It then generates synthesizable VHDL for deployment on an FPGA device. To support this, we implemented the VHDL for each of the components de-

scribed in Table 3.1. The memory generator then needs only generate the interfaces around the memory subsystem components. For streaming applications, we implemented a priority arbiter, described in Chapter 5, which the memory generator uses to connect the various components to a single main memory.

Chapter 4: Superoptimization of Memory Subsystems

4.1 Introduction

In this chapter we present the general method for superoptimizing memory subsystems for single-threaded applications and the outcome for several benchmark applications. This chapter is based on [122] and [123].

4.2 Method

Here we describe how one generates a superoptimized memory subsystem for a single-threaded application. This process involves several steps. First we require a memory address trace from the application. This trace allows us to simulate the performance of the application with different memory subsystems. Next, we perform the superoptimization, which involves generating proposal memory subsystems and simulating them to determine their performance. Finally, we generate the memory subsystem to be used in the application.

4.2.1 Address Traces

In order to evaluate the performance of a particular memory subsystem for an application, we use address traces. There are many ways of obtaining an address trace for an application. Here we consider three distinct methods:

- traces gathered from applications run on a CPU,
- traces gathered from synthetic kernels, and
- traces generated from ScalaPipe [120] applications.

To gather the address traces for applications running on a CPU, we use a modified version of the Valgrind [84] *lackey* tool. This allows us to obtain concise address traces for applications that contain only data accesses (reads, writes, and modifies). We ignore instruction accesses since the instructions would likely be stored in a separate memory, such as a read-only memory (ROM) or in the FPGA or ASIC logic itself. We use this method of address trace acquisition for existing applications, such as applications in the MiBench benchmark suite [47].

A synthetic kernel is a kernel where we use an application to generate an address trace directly rather than performing the computation. For example, the address trace for matrix-matrix multiply can be generated without the need to actually perform the multiplication, as can many others. An advantage of this method is that the address trace can be computed on-the-fly as it is being simulated with little overhead, which saves us from storing large traces.

Finally, for applications implemented in ScalaPipe, we can generate traces automatically. By extending ScalaPipe with the ability to instrument generated applications, address traces

can be gathered by running the application on a CPU target. Since ScalaPipe supports high-level synthesis, the application can then be retargeted to an FPGA device.

For now, we ignore the notion of processing time in the trace for all of the address traces. This is because our focus is exclusively on memory performance. Because there is no notion of processing time, however, certain memory subsystem components, such as prefetchers, are less likely to be useful. Introducing processing time is possible, but to do so would require a specific implementation of the application, which would make the results less general.

All of the address traces contain virtual (instead of physical) addresses and are gathered for 32-bit versions of the benchmark applications. To evaluate a general-purpose memory subsystem, the physical addresses are important since some levels of cache use physical addresses to avoid flushing the whole cache when context switching. However, we note that our memory subsystems are specific to an application and, therefore, using virtual addresses is appropriate. Further, in embedded devices as well as ASICs and FPGAs, it is often the case that only a single application is executed.

4.2.2 Simulation

To evaluate the performance of the memory subsystems proposed by the superoptimizer, we use the custom memory simulator described in Section 3.2. As previously mentioned, 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 total memory access time. Note that cache misses would not provide enough information to the superoptimizer for it to decide between a single level and a multi-level cache, for example, and simply using memory access time is insufficient for deciding how to divide up the memory resources among multiple memory subsystems. Finally, the

Parameter	Description	Value
Frequency	DRAM I/O frequency	400 MHz
CAS	Cycles to select a column	5
RCD	Cycles from open to access	5
RP	Cycles required for precharge	5
Page size	Size of a page in bytes	1024
Page count	Number of pages per bank	65536
Width	Channel width in bytes	8
Burst size	Number of columns per access	4
Page mode	Open or closed page mode	open
DDR	Double data rate	true

Table 4.1: Main Memory Parameters

simulator must be fast enough to simulate large traces many thousands of repetitions in a reasonable amount of time.

The memory subsystem superoptimizer supports seven distinct subsystem components, described in detail in Section 3.2. However, adding additional components is simply a matter of adding a synthesizable HDL model of the component and a simulation model for the memory subsystem simulator and superoptimizer. Likewise, additional parameters can be added to the existing components. Unfortunately, adding additional components or parameters can make the superoptimization process take longer since more steps will be required to explore the search space.

The communication between each of the memory components as well as the communication between the application and main memory is performed using 4-byte words. The bytes within the word are selected using a 4-bit mask to allow byte-addressing. The address bus is 30 bits, providing a 32-bit address space.

For the results presented here, the main memory is assumed to be a DRAM device. As is the case with the memory subsystems, it is possible to model main memories with other

properties if required. For our purposes, we consider a DDR3-800D memory, whose properties are shown in Table 4.1.

We target both a FPGA platform and an ASIC. For the FPGA platform, we target a Xilinx [130] Virtex-7 with a 250 MHz clock. We assume there are 64 BRAMs available for the deployment of our custom memory subsystems. For the ASIC, we target a 45nm process and assume that there is $1mm^2$ available for the deployment of our custom memory subsystems. We assume a clock frequency of 1 GHz for the ASIC target.

4.2.3 Optimization

To guide the optimization process, we use a variant of *threshold acceptance* [34] called *old bachelor acceptance* [51]. Old bachelor acceptance is a Markov-chain Monte-Carlo (MCMC) stochastic hill-climbing technique similar to simulated annealing [63]. Old bachelor acceptance provides a compromise between search space exploration and hill climbing. Thus, although we may not get the best possible memory subsystem with this technique, we do get fairly good results in much less time than it would take to perform an exhaustive search.

We use old bachelor acceptance because of its ability to avoid local optima without the need to select an overly high initial threshold or slow cooling schedule. With simulated annealing, the guarantee of discovering the optimal result relies on a very slow or adaptive cooling schedule [38]. Thus, the use of simulated annealing is impractical even if we want to be guaranteed an optimal result. If we are willing to drop the optimality requirement, we can select a cooling schedule for simulated annealing that would allow the search to converge more quickly, however, selecting such a cooling schedule requires a trade off between search time and the quality of the result. With old bachelor acceptance, we can use a more aggressive cooling schedule since the threshold can increase. There are other metaheuristics

that could be explored to reduce the search time, however, here we consider only old bachelor acceptance.

Using stochastic hill-climbing, one typically selects an initial state, $s_t = s_0$, and then generates a *proposal* state, s^*, in the neighborhood of the current state. The state is then either accepted, becoming s_{t+1}, or rejected. With threshold acceptance, the difference in cost between the current state, s_t, and the proposal state, s^*, is compared to a threshold, T_t, to determine if the proposal state should be accepted. Thus, we get the following expression for determining the next state:

$$s_{t+1} = \begin{cases} s^* & \text{if } c(s^*) < c(s_t) + T_t \\ s_t & \text{otherwise} \end{cases}$$

For our purposes, the state is a candidate memory subsystem and the cost function, $c(\cdot)$, is the total access time in cycles that the application will experience from memory accesses.

With threshold acceptance, the threshold is initialized to some relatively high value, $T_t = T_0$. The threshold is then lowered according a cooling schedule. The recommended schedule in [34] is $T_{t+1} = T_t - \Delta T_t$ where $\Delta \in (0, 1)$. Old bachelor acceptance generalizes this, allowing the threshold to be lowered when a state is accepted and raised when a state is not accepted. This allows the algorithm to escape areas of local optimality more easily. For our experiments, we used the following schedule:

$$T_{t+1} = \begin{cases} T_t - \Delta T_t & \text{if } c(s^*) < c(s_t) + T_t \\ T_t + \Delta T_t & \text{otherwise} \end{cases}$$

Because the evaluation of a state involves simulating a memory subsystem for an address trace, each state evaluation can take several minutes or even longer depending on the size of the trace. Further, to discover a good memory subsystem, the total number of states visited can be large, which can make the optimization process take a prohibitively long time.

To reduce the time required for superoptimization, we employ two techniques to speed up the process. First, we memoize the results of each state evaluation so that when revisiting a state we do not need to simulate the memory trace again. The second improvement is that we allow multiple superoptimization processes to run simultaneously sharing results using a database, thereby allowing us to exploit multiple processor cores.

4.2.4 Neighborhood Generation

Our memory subsystem optimizer is capable of proposing candidate memory subsystems comprised of the structures shown in Table 3.1. These components can be combined in arbitrary ways leading to a huge search space limited only by the constraints.

For the FPGA target, the constraints include the minimum clock frequency and the maximum number of block RAMs (BRAMs) for the memory subsystem. BRAMs are fast on-chip memories that have a configurable aspect ratio. For the ASIC target, the constraint is the area as reported from the CACTI tool [113].

Given a state, s_t, we compute a proposal state s^* by performing one of the following actions:

1. Insert a new memory component to a random position,
2. Remove a memory component from a random position, or
3. Change a parameter of the memory component at a random position.

With MCMC algorithms such as simulated annealing and threshold acceptance, it is necessary that the generated proposal states be *ergodic*. Ergodicity means that it is possible to reach every state from any given state in a finite number of steps. Obviously this property is necessary since if one hopes to reach an optimal solution, one must be able to get to that solution via some number of steps from a non-optimal solution. It is easy to see that the proposal generation process described above is ergodic as actions 1 and 2 are capable of canceling each other and action 3 can cancel itself.

To ensure that any discovered memory subsystem is valid, we reject any memory subsystem that exceeds the constraints. However, there are other ways a memory subsystem may be invalid. First, because we support splitting between memory components by address, any address transformation occurring in a split must be inverted before leaving the split. To handle this, we always insert (or remove) both the transform and its inverse when inserting (or removing) an address transformation.

Another situation that can lead to an invalid memory subsystem is when a complex memory subsystem prevents the subsystem from achieving the required clock frequency on the FPGA device. Note that for an ASIC device we increase the number of cycles required to access the memory component. Although we synthesize each component for the FPGA target separately to prevent this, it is still possible that a combination of components prevent the complete memory subsystem from achieving the required clock frequency.

To prevent the optimizer from generating a memory subsystem that is unable to run at the required clock frequency, the optimizer keeps a rough estimate on the longest combinational path and prevents the path from becoming too long. Nevertheless, it is still possible that a particular superoptimized memory subsystem may not achieve the required clock frequency. Therefore, for the FPGA results, we synthesize the superoptimized memory subsystems to validate them.

4.2.5 Offset Selection Heuristic

Because the search space is so large, arbitrarily selecting addresses to segment the address space in a split component can be problematic. Therefore, rather than proposing arbitrary addresses for split offsets, we restrict the set of addresses to values that actually exist in the address trace. We do this by recording the address ranges that are used during the first evaluation of the trace for the initial state. To further improve these results, the addresses we generate are weighted such that those addresses at the ends of address ranges are more likely to be selected.

Given an address range of length n that starts at a, addresses used for splits are selected according to the following algorithm:

$$A(a,n) = \begin{cases} a & \text{w.p. } 1/8 \\ a+n-1 & \text{w.p. } 1/8 \\ A(a, \lceil n/2 \rceil) & \text{w.p. } 3/8 \\ A(a+\lfloor n/2 \rfloor, \lceil n/2 \rceil) & \text{w.p. } 3/8 \end{cases}$$

Here *w.p.* stands for "with probability". Thus, there is a 12.5% chance of selecting the first address in the range, a 12.5% chance of selecting the last address, and a 75% chance of selecting an address between these two extremes.

4.2.6 Model Validation

To validate the simulation model used during the optimization process, our optimizer generates synthesizable VHDL that has the characteristics shown in Table 3.1. By synthesizing

the VHDL, we can ensure that the discovered memory subsystem is able to run at the required frequency and fit on our target device. The synthesis targets a Xilinx [130] Virtex-7 running at 250 MHz.

4.3 Benchmarks

We use a collection of six benchmarks from the MiBench benchmark suite [47] as well as four synthetic kernels for evaluation purposes. The MiBench benchmark suite contains single-threaded benchmarks for the embedded space that target a variety of application areas. For some benchmarks, the MiBench suite contains large and small versions. We chose the large version in the interest of obtaining larger memory traces.

The locally developed synthetic kernels include a kernel that performs random lookups in a hash table (`hash`), a kernel that performs matrix-matrix multiply (`mm`), a kernel that inserts and then removes items from a binary heap (`heap`), and a kernel that sorts an array of integers using the Quicksort algorithm [50] (`qsort`). Rather than implement an application to perform these operations and use Valgrind to capture the address trace, the addresses traces for these kernels are generated directly during a simulation run, which allows us to avoid processing large trace files for the kernels.

Because we are superoptimizing the memory subsystem, the amount of memory accessed by each benchmark is important. If a particular benchmark accesses less memory than is available to the on-chip memory subsystem, then it should be possible to have all memory accesses occur in on-chip memory, though such a design may require clever address transformations. A graph of the total working-set size for each benchmark is shown in Figure 4.1.

In Figure 4.1, we see that there are two benchmarks, `bitcount` and `dijkstra`, that are small enough that all memory accesses could be mapped into 64 BRAMs, which is 2,359,296

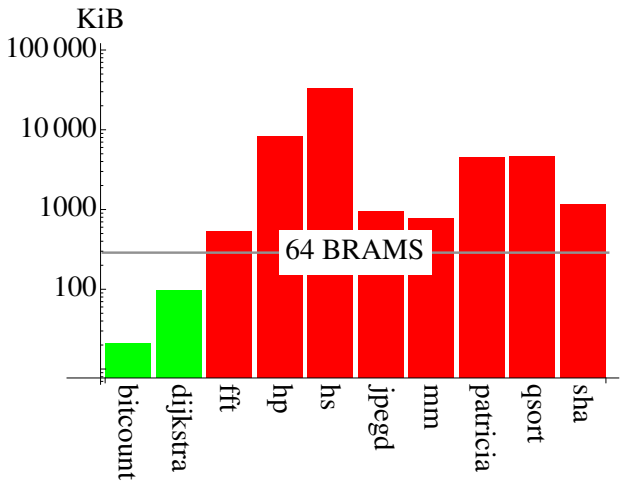


Figure 4.1: Working-Set Sizes

bits, or 294,912 bytes. All of the other benchmarks are too large to fit completely within 64 BRAMs, which is the constraint on BRAMs we consider for the FPGA target.

For the 45nm ASIC process with an area constraint of $1mm^2$, we can store a total of 379,392 bytes in a scratchpad according to our CACTI model. This means that, as with the FPGA, both the `bitcount` and `dijkstra` benchmarks are small enough to be mapped into a scratchpad, but all of the remaining benchmarks access too much memory.

4.4 Minimizing Total Access Time

Here we present the results of superoptimizing the memory subsystem to minimize total access time. To evaluate the performance of our superoptimized memory subsystems, we compare the performance of the superoptimized memory subsystems against a baseline cache. For our baseline cache, we selected a cache that closely resembles the data cache in a Raspberry Pi [95]. This is a 64 KiB, 4-way set-associative write-back cache with 32-byte lines and a PLRU replacement policy. The FPGA implementation of this cache uses 16 BRAMs and

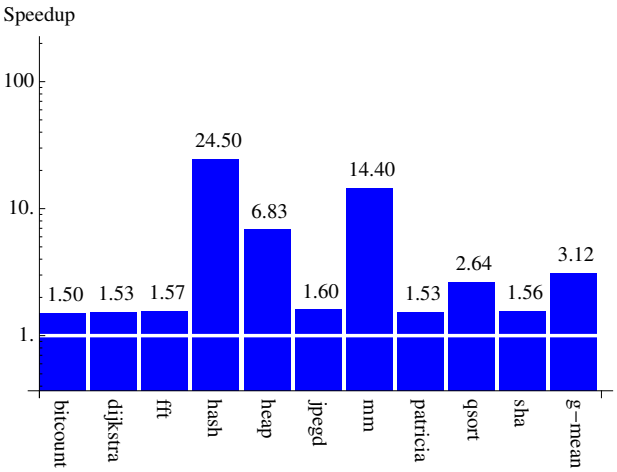


Figure 4.2: Best-case FPGA Speedup

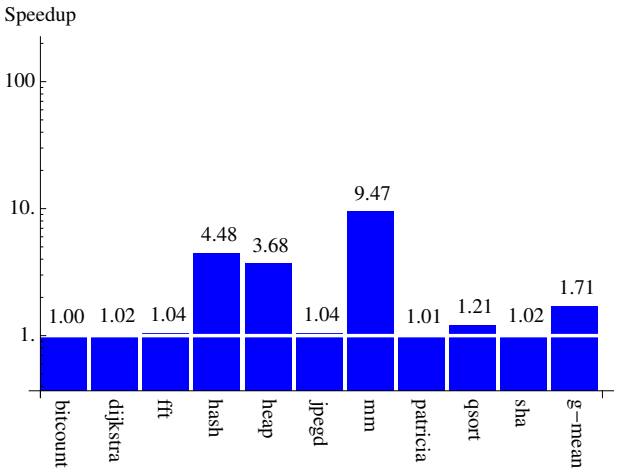


Figure 4.3: Realized FPGA Speedup

meets our 250 MHz target frequency. According to CACTI, the 45nm ASIC implementation is 0.18mm^2 with a 1-cycle access time and a 3-cycle cycle time.

4.4.1 FPGA Results

For the first set of experiments, we target a Xilinx [130] Virtex-7 with a target frequency of 250 MHz and a constraint of 64 BRAMs maximum. On the Virtex-7 part, each BRAM has a base aspect ratio of 512 bits by 72 bits, or 36,864 bits. The main memory is assumed to be the DDR3 device whose properties are shown in Table 4.1.

The first question we attempt to answer is: how much better might we make the memory subsystem than the baseline cache? To determine this, we compare the performance of each benchmark to a "best-case" access time. For the best-case access time, we assume that all memory accesses hit in the fastest memory component available for each of our targets. For the FPGA target, this means that all accesses hit in a scratchpad and, therefore, take two cycles to complete. This best-case speedup for our benchmarks running on the FPGA target is shown in Figure 4.2.

The *g-mean* bar in Figure 4.2 represents the geometric mean. Assuming that we could somehow arrange for all of the memory accesses to hit in the scratchpad we would get a 3.12× speedup over the baseline cache for the FPGA target. Note that, in reality, such a speedup is not possible since we do not have enough resources available to make all of the accesses hit in a scratchpad.

Figure 4.3 shows the speedup that the superoptimized memory subsystem provides over the baseline memory subsystem. Across the set of benchmark applications, the performance gain varies from very little to over 9× with a geometric mean speedup of 1.71×.

Although some of the results are not much better than the baseline memory subsystem, we note that for all of the benchmarks there was some improvement, though less than 1% in a few cases. There are a few benchmarks, however, that exhibit substantial performance gain. The matrix-matrix multiply kernel shows the best speedup of over 9×. Because the main memory is not much slower than the cache structures running on a 250 MHz FPGA fabric, we do not anticipate substantial gains for all of the applications (see Figure 4.2). A number of the discovered memory subsystems are, however, worth considering in more detail.

The first interesting memory subsystem we consider is the superoptimized memory subsystem for the `hash` benchmark, shown in Figure 4.4a. The `hash` benchmark performs random probes into a `hash` table containing 8,388,608 entries, each 4-bytes. This type of access pattern causes problems for caches due to the lack of locality. In Figure 4.4a, memory accesses enter the top and accesses to main memory come out the bottom. There are two address transformations and a 262,144-byte scratchpad. The first address transformation toggles a bit of the address. The transformed address then enters the scratchpad. The second transformation reverses the first transformation so that the addresses remain unchanged as they enter the main memory (recall that address transformations are always inserted and removed in pairs).

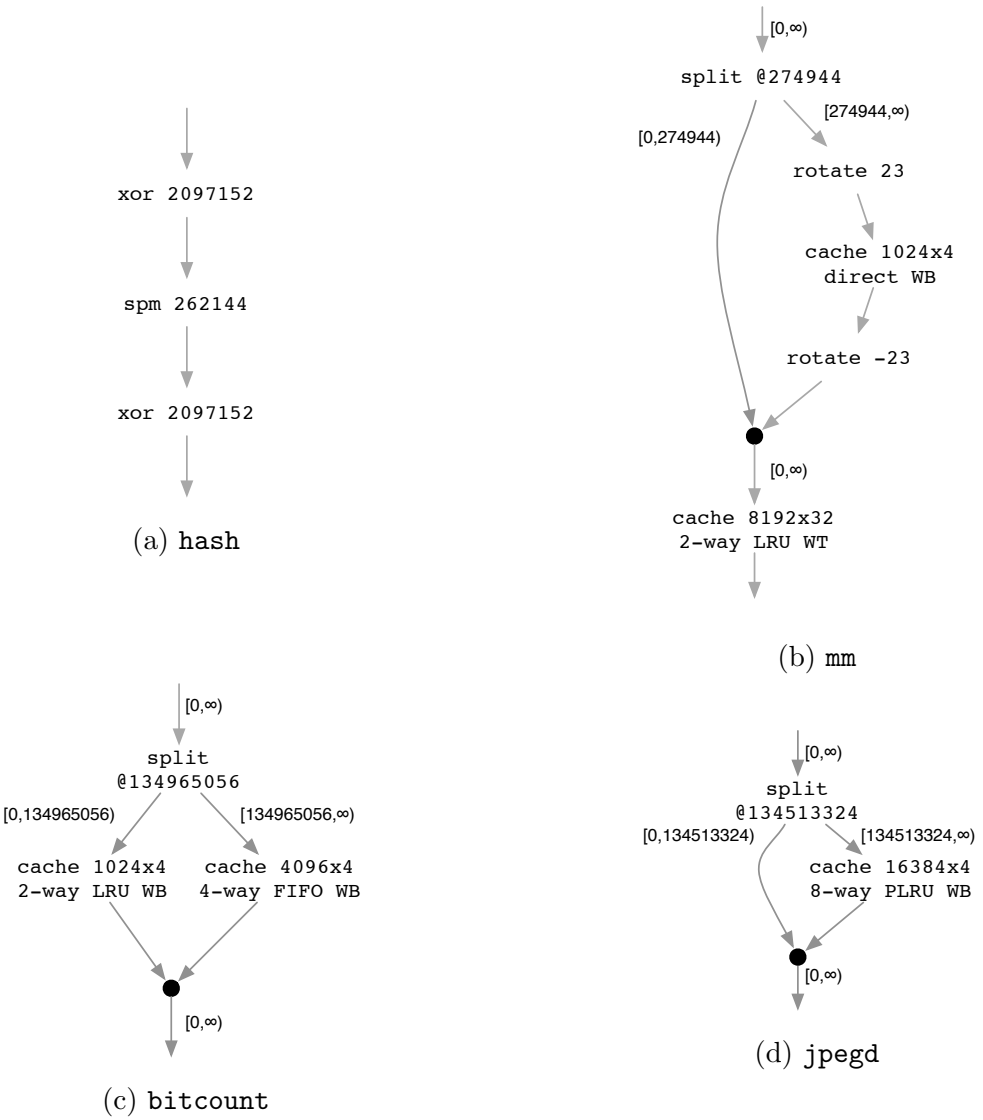


Figure 4.4: Superoptimized Memory Subsystems for the FPGA Target

The reason that the address transformation is beneficial for the `hash` benchmark is due to the random accesses to the hash table being slightly unbalanced. Removing the address transformation results in a very slight decrease in performance. If we remove the scratchpad completely, there is again only a slight decrease in performance. Here we note that the speedup is primarily due to the removal of the cache, which serves only to cause overhead when there is no locality. The scratchpad speeds up some of the accesses, but only a small fraction.

Another interesting memory subsystem, which also provides the greatest performance improvement, is discovered for `mm`: the matrix-matrix multiply benchmark. This benchmark performs a matrix-matrix multiply using the naive $O(n^3)$ algorithm with 256-by-256 matrices. Each element of the matrix is 4 bytes. The superoptimized memory subsystem for this benchmark is shown in Figure 4.4b. In the superoptimized memory subsystem for the `mm` benchmark, memory accesses enter the top and are then split, with accesses below address 274944 going directly to a 262,144-byte cache at the bottom of Figure 4.4b and accesses to addresses above and including 274944 going to a separate memory subsystem before going to the 262,144-byte cache. For accesses to addresses above and including 274944, first the bits of the address that select the word are rotated left by 23 bits. The accesses then enter a 4,096-byte, direct-mapped cache, and finally, the address is rotated right by 23 bits before entering the larger cache.

To understand why the memory subsystem for the `mm` benchmark provides such good performance, we consider the way the memory is organized for the benchmark. There are 3 matrices: two sources and a destination. The first source matrix, which is accessed in row-major order, is stored in addresses 0 through 262140. The second source matrix, which is accessed in column-major order, is stored at addresses 262144 through 524284. Finally, the destination matrix is stored at addresses 524288 through 786428.

With this memory organization in mind, we note that the address split moves most accesses for the second source matrix as well as the destination matrix into a separate memory subsystem. Within this subsystem, the addresses are transformed and then routed to a cache. Given that the second source matrix is accessed in column-major order, for the first column, we access 00040000_{16}, 00040400_{16}, ... $0007FC00_{16}$ for the first column, then 00040004_{16}, 00040404_{16}, ... $0007FC04_{16}$ for the second column, and so on. However, after the split and address transformation, the addresses from the perspective of the 1024 entry cache look about like this: 00000000_{16}, 00000008_{16}, ... $00000FF8_{16}$ for the first column, 01000000_{16}, 01000008_{16}, ... $01000FF8_{16}$ for the second column, and so on. The result is that each column of the matrix is cached and can be reused 256 times before the next column is required.

Note that due to the layout of the matrices, one would expect that the ideal address for the split would be 262144 instead of 274944. Indeed, changing the split address results in a 0.46% improvement in performance. Thus, running the superoptimizer longer would likely result in an even better memory subsystem. Further, this implies that there may be better ways to propose offsets for splits.

A final observation about the memory subsystem for the `mm` benchmark is the large cache after the split. This cache has 32-byte cache lines, which allows it to prefetch values for the source matrix. Also, the cache is write-through rather than write-back, which prevents cache pollution due to writes to the destination matrix.

The memory subsystem discovered for the `bitcount` benchmark is shown in Figure 4.4c. This memory subsystem only provides a small performance improvement over the baseline (a speedup of less than 1%), but it also uses fewer block RAMs than the baseline memory subsystem (9 instead of 16). This feat is accomplished by splitting the address space between two caches. The first cache handles accesses to heap allocations whereas the second cache

handles accesses to the stack. This type of split is common for the benchmarks that have accesses to a separate stack and heap.

Finally, we consider the memory subsystem for the `jpegd` benchmark, shown in Figure 4.4d. For the `jpegd` benchmark, the superoptimizer selected a split memory subsystem where only memory accesses to addresses 134513324 and higher go to a cache. This causes accesses to the program stack to be cached, but not accesses to heap allocations.

Of the superoptimized memory subsystems for the FPGA target, none contained only a single-level cache component. Five of the memory subsystems contained splits (`bitcount`, `fft`, `jpegd`, `mm`, and `sha`), five contained scratchpads (`dijkstra`, `hash`, `heap`, `patricia`, and `qsort`), and five contained address transformations (`dijkstra`, `hash`, `mm`, `patricia`, and `qsort`). Further, all of the superoptimized memory subsystems performed better than the baseline memory subsystem, even if only marginally better in some cases.

4.4.2 ASIC Results

For the next set of experiments, we target a 45 nm ASIC process running at 1 GHz. Using CACTI [113] for area and timing results, we constrain the area to $1mm^2$. As with the FPGA target, the main memory is assumed to be the DDR3 device whose properties are shown in Table 4.1.

The best-case speedup for the ASIC target is shown in Figure 4.5. For the ASIC target, we assume that, in the best case, all memory accesses hit a scratchpad with a 1-cycle access time and cycle time. Here we see that the geometric-mean best-case speedup is 17×. As in the FPGA case, it is not necessarily possible to achieve such a speedup.

Figure 4.6 shows the speedup that the optimized memory subsystem provides over the baseline memory subsystem. The geometric mean speedup is 6.52×. The superoptimizer is able

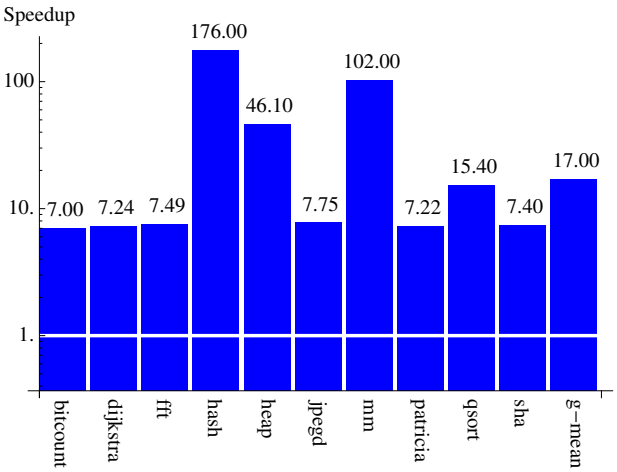


Figure 4.5: Best-case ASIC Speedup

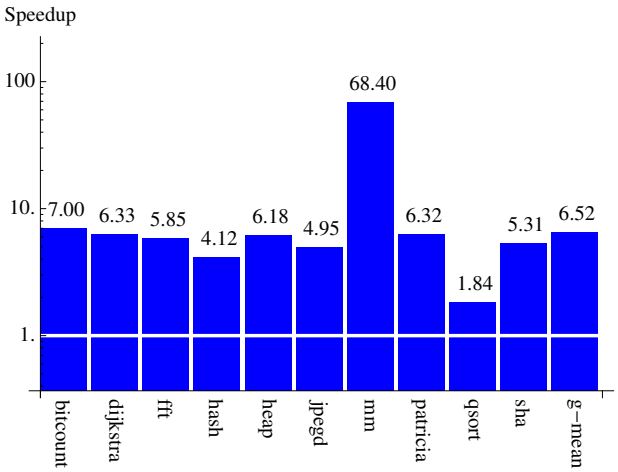
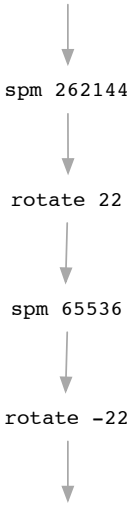


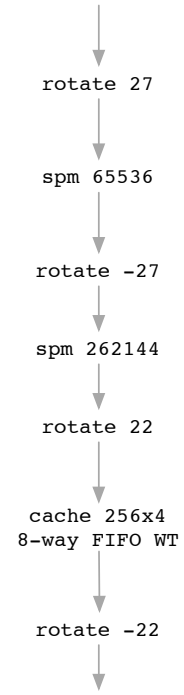
Figure 4.6: Realized ASIC Speedup

to get more impressive speedups for the ASIC than the FPGA for two reasons. First, the ASIC is assumed to be running at a higher clock frequency than the FPGA (1 GHz versus 250 MHz), making a miss in the memory subsystem have a greater impact. Second, there are more trade-offs for the ASIC memory components. In particular, when targeting an ASIC, the optimizer uses the access time and cycle time results from CACTI rather than using a fixed access time and cycle time as is done for the FPGA.

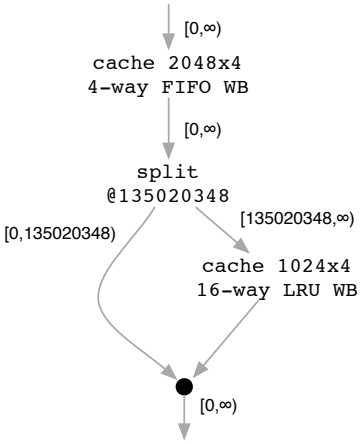
The greatest increase in performance is again seen for the `mm` benchmark, whose memory subsystem is shown in Figure 4.7b. This memory subsystem has two sets of address rotations. The rotation by 27 bits causes every eighth entry of the first source matrix for 16384 entries to be stored in the first scratchpad, which has a cycle time of 1 cycle. Another 65536 entries of the first source matrix are stored in the second scratchpad, which has a cycle time of 3 cycles. Finally, the second set of rotations causes columns of the second source matrix to be cached in a way similar to the memory subsystem for the FPGA. Although the first address rotation may seem unnecessary, by reducing conflict misses in the cache, it actually improves the performance of the memory subsystem.



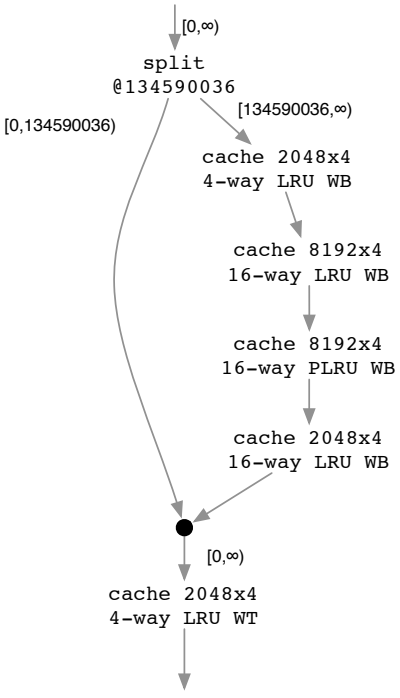
(a) `hash`



(b) `mm`



(c) `bitcount`



(d) `jpegd`

Figure 4.7: Superoptimized Memory Subsystems for the ASIC Target

The memory subsystem for the `hash` benchmark targeting the ASIC is shown in Figure 4.7a. As is the case with the `mm` benchmark, the subsystem for the `hash` benchmark is similar to the subsystem for the FPGA. However, rather than an xor transform, this subsystem uses a rotate. In addition, this subsystem incorporates two scratchpads instead of one.

The memory subsystem discovered for the `bitcount` benchmark, shown in Figure 4.7c, is similar to the memory subsystem discovered for the `bitcount` benchmark for the FPGA target, shown in Figure 4.4c. Note that the split offset is only slightly different. However, here we have a cache before the split rather than on the left side of the split.

The last memory subsystem we consider in detail is the memory subsystem for the `jpegd` benchmark, shown in Figure 4.7d. This memory subsystem is one of the most complex memory subsystems discovered. The split causes access to the memory in the stack space to be mapped to a 4-level cache. Finally, accesses to both the stack and heap are backed by a smaller cache. The four levels of cache in the split each have slightly different properties and removing any one of the caches causes a decrease in performance. Having separate, smaller caches such as this can be beneficial since smaller caches are faster than larger caches.

As is the case with the FPGA target, none of the superoptimized memory subsystems for the ASIC target contained only a single-level cache component. Four of the memory subsystems contained splits (`bitcount`, `jpegd`, `patricia`, and `sha`), six contained scratchpads (`dijkstra`, `fft`, `hash`, `heap`, `mm`, `qsort`) and six contained address transformations (`dijkstra`, `fft`, `hash`, `heap`, `mm`, and `qsort`). Further, like the FPGA target, all of the superoptimized memory subsystems performed better than the baseline memory subsystem.

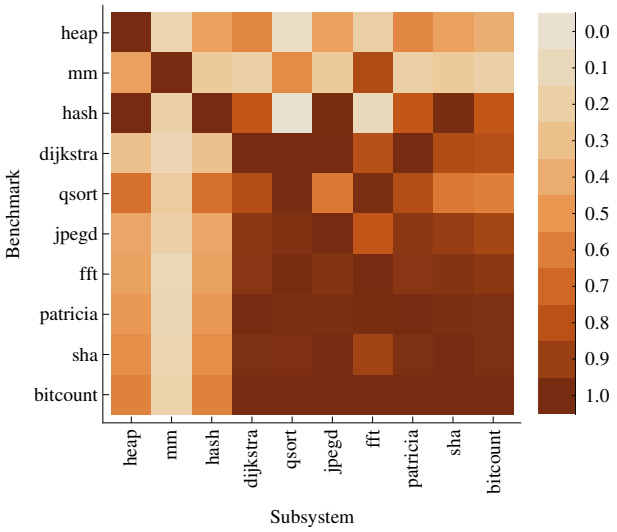


Figure 4.8: FPGA Subsystem Specificity

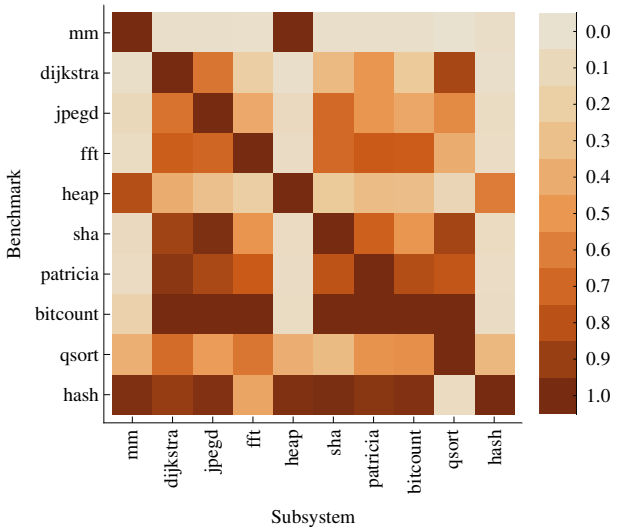


Figure 4.9: ASIC Subsystem Specificity

4.4.3 Memory Subsystem Specificity

Finally, we consider how specific each of the memory subsystems is to the application for which the subsystem was superoptimized. Figure 4.8 shows a heat map comparing the results of running each of the 10 benchmarks with each of the 10 superoptimized memory subsystems for the FPGA target. The results are computed by dividing the total access time of each benchmark running with each memory subsystem by the total access time of the benchmark running with the memory subsystem that was superoptimized for that benchmark. In the figure, darker colors represent better performance.

In Figure 4.8, we see that the `mm` and `heap` benchmarks appear to run well only on the memory subsystems that are superoptimized for them. For the `mm` benchmark, the performance improvement from the rotate in the memory subsystem is significant enough to prevent any of the other memory subsystems from approaching the performance of the `mm` memory subsystem. The `heap` benchmark contains only a scratchpad, which causes accesses to the start of the heap, which are most frequent, to be fast. However, such a structure

is suboptimal for the other benchmarks, though the `hash` benchmark performs fairly well with the memory subsystem for the `heap` benchmark. In all cases, the memory subsystem that was superoptimized for a particular benchmark provides the best performance for that benchmark.

Figure 4.9 shows a heat map comparing the results of running each of the 10 benchmarks with each of the 10 superoptimized memory subsystems for the ASIC target. As is the case with the FPGA results, the benchmarks all perform best with the superoptimized memory subsystem for the particular benchmark. In fact, the results are more specific for the ASIC target than for the FPGA target, which is likely due to the fact that the ASIC target runs faster and has a more complex search space.

Given that the superoptimized memory subsystems are specific to the benchmark for which they were superoptimized, we note that the memory subsystem may further be specific to a particular run of the benchmark. To investigate this, we used a different input data set of the same size for each of the benchmarks for the ASIC target. For example, for the `jpegd` benchmark, a different input image of the same dimensions as the original was chosen. A comparison of the speedups over the baseline memory subsystem for the original data set and the new data set is shown in Figure 4.10.

In Figure 4.10, the lighter bars (on the left) show the speedup of the superoptimized memory subsystem over the baseline memory subsystem for the original data set and the darker bars (on the right) show the speedup for the modified data set. For many of the benchmarks there is little or no difference and in one case (`dijkstra`), the speedup actually improved. Overall, the geometric mean dropped from 6.43× to 6.27×. Although its impossible to draw anything conclusive from these results, it appears that the effects of over-fitting are minimal.

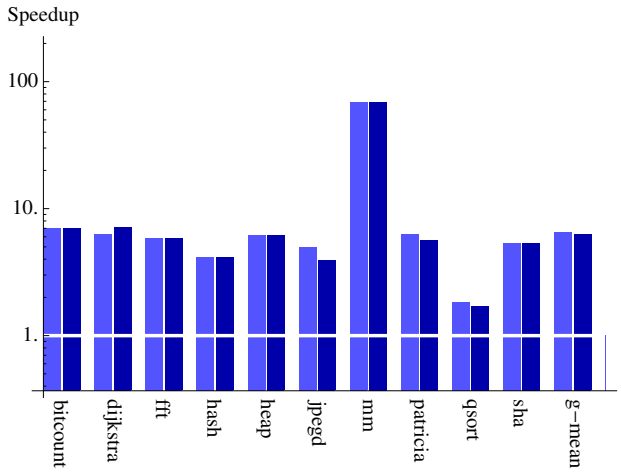


Figure 4.10: Speedup with Different Inputs

4.5 Minimizing Writes

Until now we have been focused on reducing total memory access time. However, as previously noted, the superoptimization technique is generic and, therefore, can be used to optimize for other objectives. Here we investigate minimizing the number of writes to main memory.

4.5.1 Motivation

Although DRAM is the most popular choice for main memory in modern computer systems today [98], there are several disadvantages to DRAM technology leading researchers to seek other technologies. Two such problems with DRAM include its volatility and scaling issues. Because DRAM is volatile, meaning it requires periodic refresh, DRAM can be power-hungry since it requires power just to retain information. This is particularly apparent when used in a setting with infrequent main memory accesses. However, when used in a setting with frequent memory accesses, the refresh requirement for a large DRAM can greatly reduce application performance [109]. As far as DRAM scaling is concerned, there are significant

challenges to scaling down DRAM cells [74] since bits are stored as charge on a capacitor, which further limits energy efficiency and performance.

Several alternative main memory technologies have been proposed, including PCM [69] and STT-RAM [64]. Although there are many possible main memory technologies that could be considered, a common theme for many proposed main memory technologies is an aversion to writes. For PCM, there is a limited write endurance, making it beneficial to avoid writes to extend the lifetime of the device. Further, on PCM devices writes are slow and energy-hungry. For STT-RAM, although writes do not limit the lifetime of the device, writes are much slower than reads and consume more energy. Therefore, avoiding writes to the main memory is a likely objective when faced with such a technology.

Because writes are often costly with respect to energy, performance, and endurance, here we seek to determine if it is possible to modify the memory subsystem to reduce the number of writes to main memory. We are particularly interested in the possibility of reducing the number of writes beyond what a memory subsystem superoptimized for performance would provide.

4.5.2 Results

To demonstrate the use of our superoptimization technique for the reduction of writes, here we present results for several of the benchmarks mentioned in Section 4.3. We target the ASIC platform described in Section 4.4.2. The cost function used to guide the superoptimization process is the total writes to main memory for the complete execution of the benchmark application. Thus, we are no longer optimizing for performance, but exclusively for a reduction in writes to main memory.

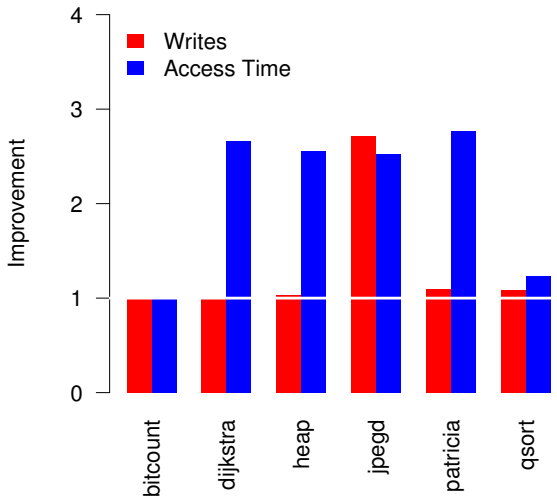
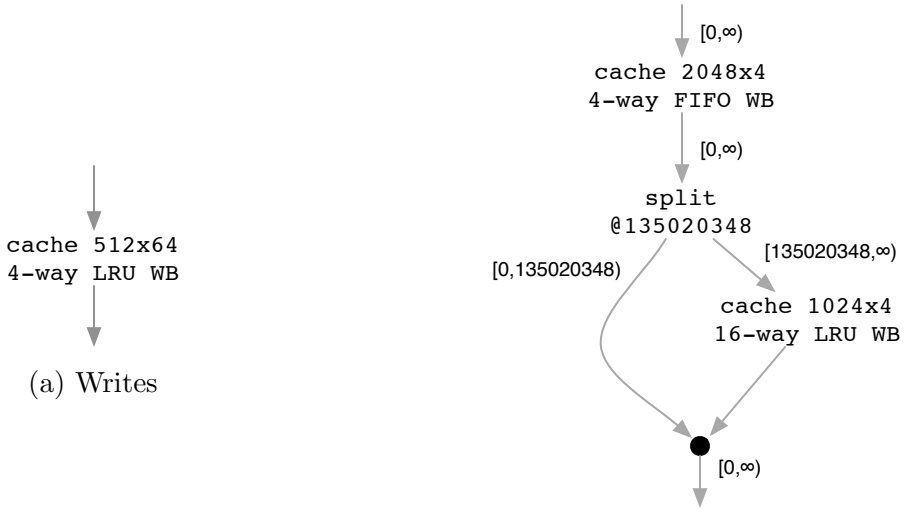


Figure 4.11: Write and Access Time Improvement

To determine if the memory subsystem superoptimized for writes is actually any better at reducing writes than a memory subsystem superoptimized for total access time, we compare each of the memory subsystems. The first column of Figure 4.11 shows the improvement that the memory subsystems superoptimized for writes have over the memory subsystems superoptimized for total access time (that is, W_t/W_w where W_w is the total number of writes to main memory when using the memory subsystem superoptimized for writes and W_t is the total number of writes to main memory when using the memory subsystem superoptimized for access time). The second column shows the improvement that the memory subsystems superoptimized for total access time have over the memory subsystems superoptimized for writes (that is, T_w/T_t, where T_t is the total access time when using the memory subsystem superoptimized for access time and T_w is the total access time when using the memory subsystem superoptimized for writes). Thus, bars above one indicate an advantage of one superoptimized memory subsystem over another.



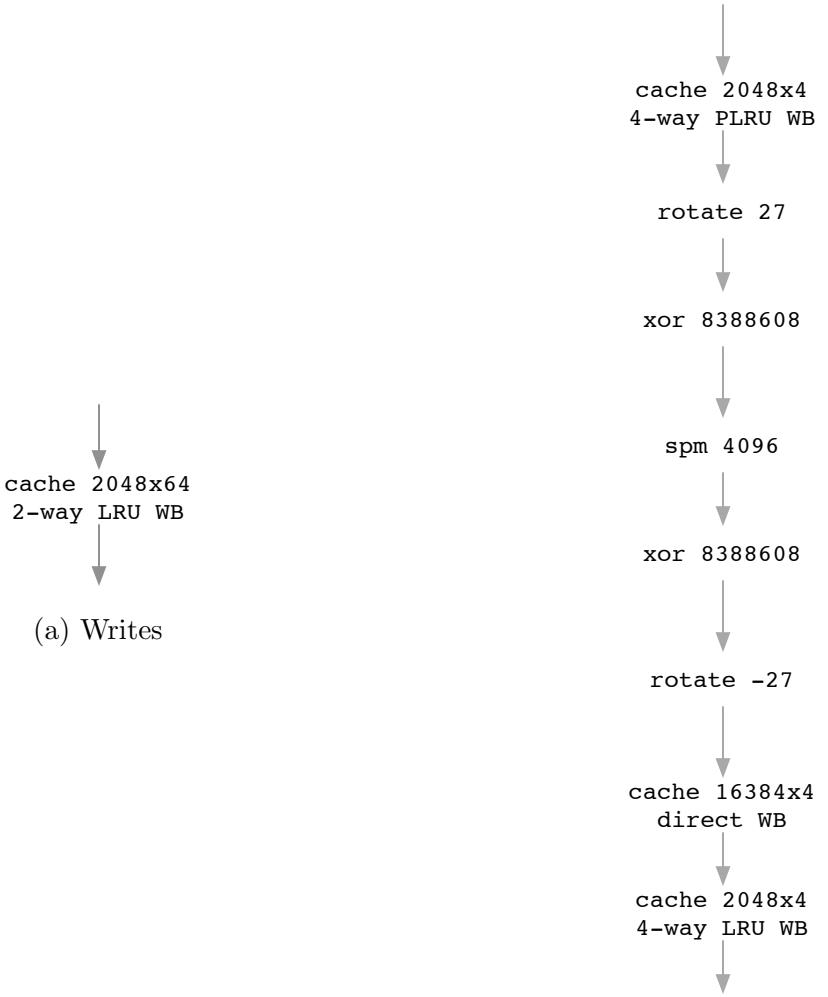
(a) Writes

(b) Access Time

Figure 4.12: Superoptimized Memory Subsystems for `bitcount`

Here we expect all bars to be at least one, indicating that the memory subsystem superoptimized for a particular objective is at least as good for that objective as a memory subsystem superoptimized for the other objective. Indeed, all bars in Figure 4.11 are one or greater. In some cases, there is little or no difference between the superoptimized memory subsystems. For example, for the `bitcount` benchmark, both memory subsystems reduce the number of writes to zero and the memory subsystem superoptimized for total access time provides only a slight improvement in total access time over the memory subsystem superoptimized for writes. However, in most cases, a different memory subsystem is able to provide the best results for either objective.

The memory subsystem superoptimized for writes for the `bitcount` benchmark is shown in Figure 4.12a and the memory subsystem superoptimized for total access time for the `bitcount` benchmark is shown in Figure 4.12b. As previously mentioned, both memory subsystems were able to reduce the number of writes to zero for this benchmark due to the small number of distinct address that are written. For this benchmark, the memory



(a) Writes

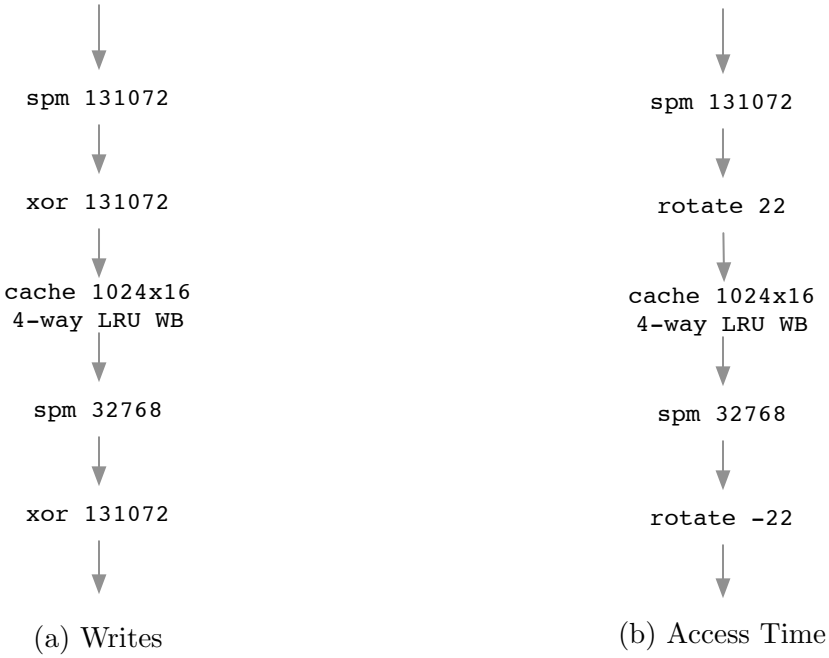
(b) Access Time

Figure 4.13: Superoptimized Memory Subsystems for `dijkstra`

subsystem superoptimized for total access time provides only a small advantage over the simpler memory subsystem that was discovered to reduce writes to main memory.

For the `dijkstra` benchmark, the memory subsystem superoptimized for writes is shown in Figure 4.13a and the memory subsystem superoptimized for total access time is shown in Figure 4.13b. As with the `bitcount` benchmark, both memory subsystems are able to reduce the number of writes to zero. However, here we note that the very unusual memory subsystem that was discovered when superoptimizing for total access time is able to provide

a much greater reduction in total access time than the memory subsystem discovered when superoptimizing for writes. Considering both subsystems were able to reduce the number of writes to zero, it is unsurprising that a simpler subsystem can be used if we do not care about total access time.

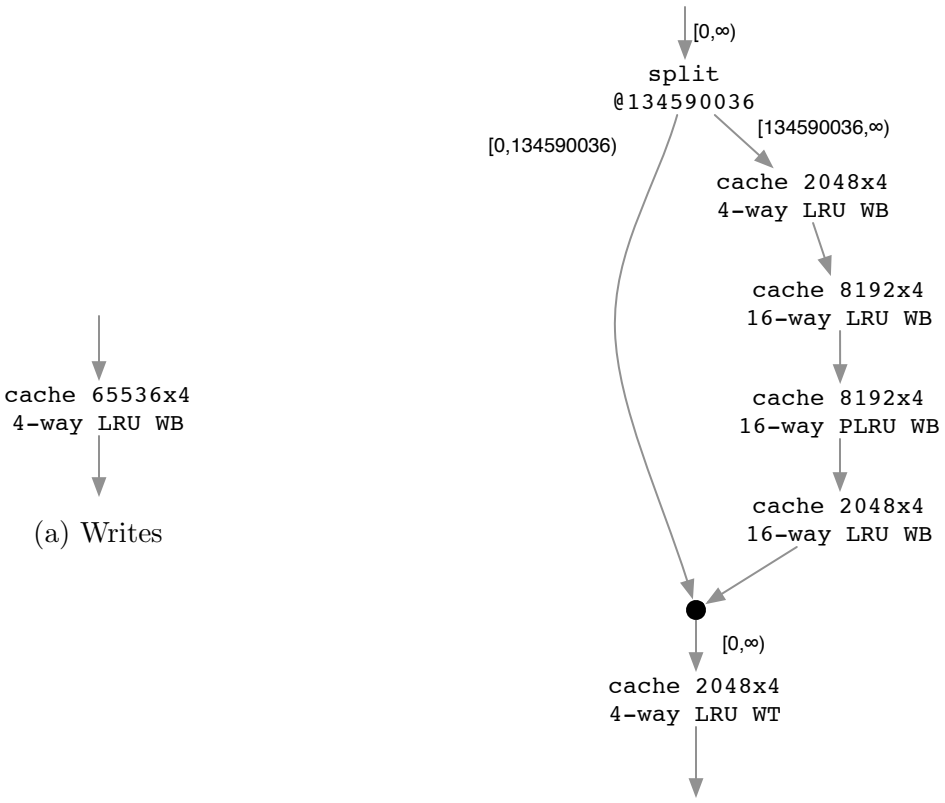


(a) Writes (b) Access Time

Figure 4.14: Superoptimized Memory Subsystems for `heap`

The next memory subsystems we consider are those superoptimized for the `heap` kernel. The memory subsystem superoptimized to minimize writes is shown in Figure 4.14a and the memory subsystem superoptimized to minimize total access time is shown in Figure 4.14b. Interestingly, these memory subsystems are very similar with the only difference being the address transformation. Despite the similar appearance, the each of the memory subsystems is able to provide a benefit over the other.

The memory subsystem superoptimized for writes for the `jpegd` benchmark is shown in Figure 4.15a and the subsystem superoptimized for access time is shown in Figure 4.15b. Again, the memory subsystem that was superoptimized to minimize total access time is



(a) Writes

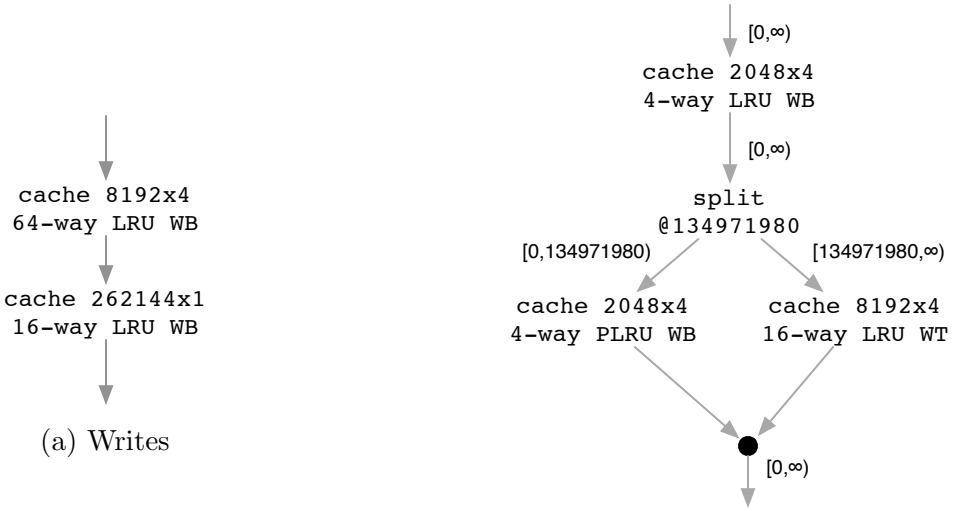
(b) Access Time

Figure 4.15: Superoptimized Memory Subsystems for `jpegd`

much more complex than the one to minimize writes. The memory subsystems for this benchmark represent the most specificity for their respective objectives of all the benchmarks attempted.

The memory subsystem superoptimized for writes for the `patricia` benchmark is shown in Figure 4.16a and the subsystem superoptimized for total access time is shown in Figure 4.16b. An interesting observation is the large and highly-associative caches that are used when minimizing writes is the objective. These caches are effective at eliminating writes, but they are quite slow.

Finally, we consider the memory subsystems for the `qsort` benchmark. Figure 4.17a shows the memory subsystem superoptimized for writes and Figure 4.17b shows the memory subsys-



(a) Writes

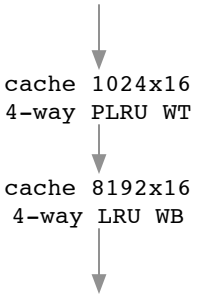
(b) Access Time

Figure 4.16: Superoptimized Memory Subsystems for `patricia`

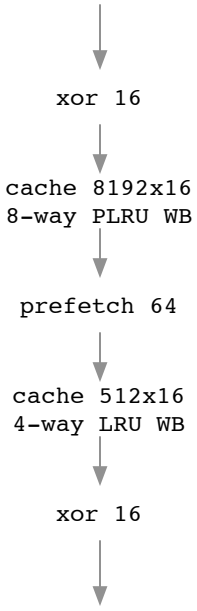
tem superoptimized for total access time. One notable difference between these subsystems is the presence of the prefetch component in the memory subsystem superoptimized for total access time.

Overall, memory subsystems superoptimized to minimize total access time appear to be capable of large reductions in total access time over memory subsystems superoptimized to minimize writes. On the other hand, while a memory subsystem superoptimized for writes is often able to reduce the number of writes compared to a memory subsystem superoptimized for total access time, the improvement is usually less pronounced. An explanation for this is that usually the total access time decreases as the number of reads and writes to main memory decrease.

Another observation is that the memory subsystems superoptimized for writes are usually simpler than those superoptimized for total access time. Although a large cache will typically eliminate writes, the large cache will likely be slow. This implies that a large cache may be



(a) Writes



(b) Access Time

Figure 4.17: Superoptimized Memory Subsystems for `qsort`

sufficient if we only care about writes, but something more exotic will likely provide better results if we want to minimize total access time.

4.6 Multi-Objective Superoptimization

Here we investigate multi-objective superoptimization. From the previous section, we note that the memory subsystems that are superoptimized to minimize total access time are fairly good at reducing the number of writes to main memory, however, the memory subsystems superoptimized to minimize writes usually do better. On the other hand, the memory subsystems that are superoptimized to minimize writes often perform poorly with respect to total access time. Thus, one might wonder if it is possible to optimize for both objectives.

We use the weighted sum method (see [75]) to combine the objective functions to minimize writes and total access time. Figure 4.18 shows the improvement possible for various objec-

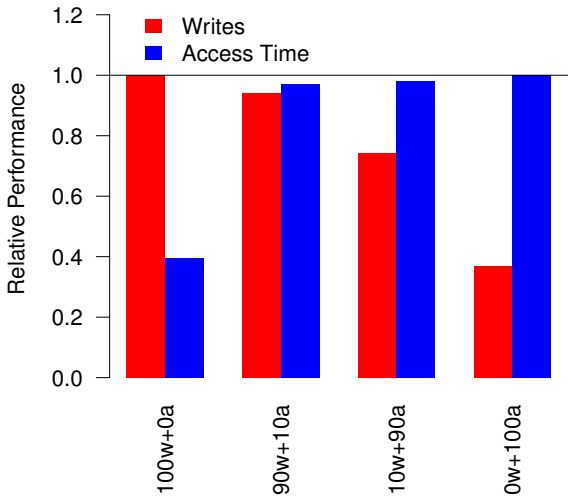
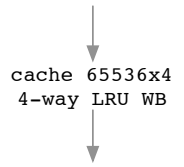


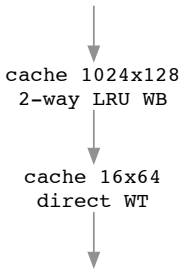
Figure 4.18: Multi-Objective Superoptimization

tives for the `jpegd` benchmark with objective weights ranging from 100%-writes, 0%-access time through 0%-writes, 100%-access time. The graph shows uses the performance relative to the best result for writes and total access time. For the bars on the left, the graph shows W_w/W_m, where W_m is the number of writes to the main memory when using the memory subsystem superoptimized for multiple objectives and W_w is the number of writes to the main memory when using the memory subsystem superoptimized to minimize writes. For the bars on the right, the graph shows T_t/T_m, where T_m is the total access time when using the memory subsystem superoptimized for multiple objectives and T_t is the total access time when using the memory subsystem superoptimized to minimize total access time. Thus, higher values (closer to 1) indicate better results.

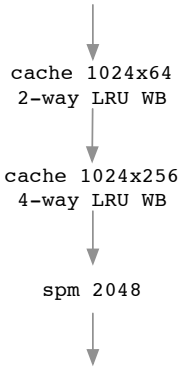
As can be seen in the graph, the largest differences in how good the memory subsystems perform for each objective occur when only a single objective is considered. When multiple objectives are considered, although there is some difference in how good the memory subsystems are, the result is very close to the best for all mixtures.



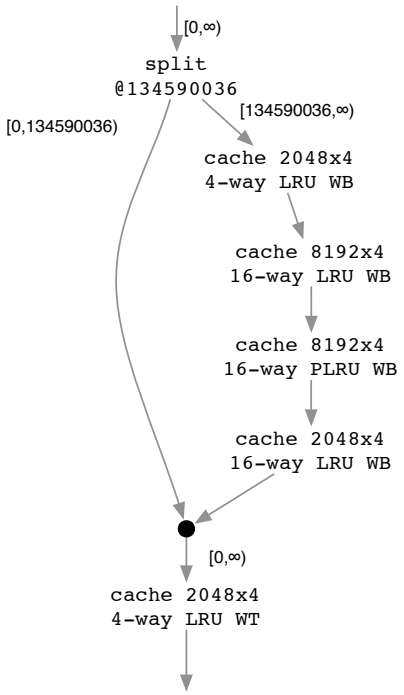
(a) $100w + 0a$



(b) $90w + 10a$



(c) $10w + 90a$



(d) $0w + 100a$

Figure 4.19: Memory Subsystems for `jpegd`

Figure 4.19 shows the memory subsystems for each mixture. When minimizing writes is most important, we see that a simple cache suffices. However, when minimizing total access time is also important, the large cache is separated into two caches, which makes sense since smaller caches are faster. Finally, when writes are no longer considered, a very complex memory subsystem is discovered, which does little to minimize writes, but provides the lowest total access time of all the memory subsystems considered.

4.7 Summary

In this chapter, we have shown that it is possible to superoptimize memory subsystems for specific applications that out-perform a general-purpose memory subsystem in terms of either performance or writes. Unlike previous work, the memory subsystems that our superoptimizer discovers can be arbitrarily complex and contain components other than simple caches. To superoptimize a memory subsystem, we use old bachelor acceptance, which is a form of threshold acceptance. We are then able to improve the discovery process by using information from the address trace.

This work targets both an FPGA as well as an ASIC process. For the FPGA target, we have validated the discovered memory subsystems by generating VHDL for each of the subsystems. The VHDL was then synthesized to ensure that the discovered memory subsystems are realizable at the required frequency. For the ASIC process, we used the CACTI [113] tool to get area and time estimates for each of the memory components.

An obvious shortcoming of the superoptimization technique presented so far is that it only works on single-threaded applications. Thus, due to the inherently parallel nature of ASIC and FPGA devices, this approach has limited applicability. Nevertheless, this technique is applicable for single-threaded applications and parallel applications in which only a single

thread accesses main memory. In the next chapter we extend the superoptimization approach to a class of parallel applications.

Chapter 5: Memory Subsystems for Streaming Applications

This chapter expands on the methods presented in Chapter 4 to superoptimize memory subsystems for streaming applications. In addition, an empirical validation of the superoptimized memory subsystems is provided for applications deployed on an FPGA target. This chapter is based on [124].

5.1 Introduction

Modern computer systems are becoming increasingly parallel [90]. This especially true of ASICs and FPGAs, which are naturally parallel devices and likely targets for superoptimized memory subsystems. Thus, here we investigate extending the notion of application-specific memory subsystems to parallel applications. In particular, we consider streaming applications, which are well-suited to heterogeneous systems consisting of general-purpose processors, FPGAs, GPUs, and other devices.

Streaming is a parallel programming paradigm where application kernels communicate over fixed communication channels. The streaming paradigm is used in systems such as ScalaPipe (described in Section 3.1) and StreamIt [112], among many others [17, 29, 45, 46, 105]. 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.

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 (low-power double date rate) 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.

5.2 Method

Given a streaming 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

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.

5.2.1 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 accurately model 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 accurately predict how long each kernel will run relative to other kernels.

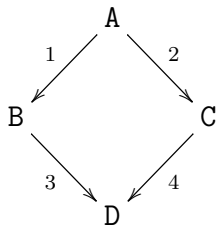


Figure 5.1: Split-Join Topology

Consider the simple streaming application topology shown in Figure 5.1. 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:

```
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 change the size of 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 5.1) and *join* kernels (such as kernel `D`) would provide a valid trace, such a trace may not give the superoptimizer sufficient freedom to resize the communication channels. 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 internally 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 first run the application 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 with a custom memory subsystem.

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 manually instrument the application to generate the required trace data. For example, an application implemented in a hardware description language (HDL), such as VHDL or Verilog, could be manually instrumented and then run in a simulator.

5.2.2 Simulation

To evaluate the performance of the memory subsystems proposed by the superoptimizer, we use the custom trace-based memory simulator described in Section 3.2. For the experiments presented here, we use the fact that the simulator reports the total number of 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 5.1, which were chosen to model closely the main memory on our experimental platform.

5.2.3 Optimization

As before, we use *old bachelor acceptance* [51] to guide the superoptimization process. Unfortunately, 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 Chapter 4, here we use a heuristic to guide the superoptimizer to spend more

Parameter	Description	Value
Frequency	DRAM I/O frequency	100 MHz
CAS	Cycles 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 5.1: Main Memory Parameters

effort exploring the memory subsystems that are most likely to benefit the application. To do this, the memory 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 memory 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 memory subsystems that can gain the most benefit from the memory subsystem.

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. Note that this resource constraint differs from the constraint used in Chapter 4 in that here we are constraining based on both slices and block RAMs whereas in Chapter 4 the only constraint was the block RAMs. The additional constraint on slices makes it easier to fit the application on the FPGA along side the memory subsystem (before we were concerned only with the memory subsystem).

In addition to the resource constraints, we put 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 the results of synthesis runs 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 not considered.

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.

5.2.4 Subsystem Generation

Once a memory subsystem has been superoptimized, we use an automatic memory subsystem generator (see Chapter 3 for details) to generate a VHDL description of the memory subsystem. This subsystem generator is capable of generating all of the memory subsystems shown in Table 3.1. Each memory subsystem has a simple SRAM-style interface with per-byte write enables. The memory subsystems are connected to the main memory using a priority arbiter capable of allowing multiple outstanding main memory requests (one for each memory subsystem).

The word size of various components in the memory subsystem need not be equivalent. 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. Note that this differs from the method used in Chapter 4. In Chapter 4 all levels of the memory subsystem used a fixed word size whereas this restriction has been lifted for these results.

5.3 Benchmarks

Following is a description of the benchmarks used to evaluate our custom memory subsystems. All of the benchmarks are implemented in ScalaPipe [120]. 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 CPUs 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 first deploy the application on processor cores to generate the address traces and then generate the application on FPGA cores for deployment with our custom memory subsystems.

Merge Sort

The first benchmark we consider is the `merge` benchmark, which is a merge sort application capable of sorting up to one million 32-bit integers. This benchmark 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 5.2. Each kernel in the pipeline sorts

sequences of elements 2× longer than the sequences from the preceding kernel by using an internal buffer to store half the elements. This sort algorithm is described in detail in [18].

Due to the memory requirements of sorting one million integers, this benchmark 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.

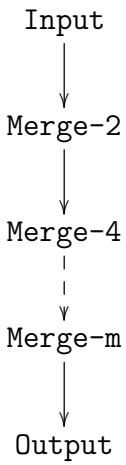


Figure 5.2: `merge` Topology

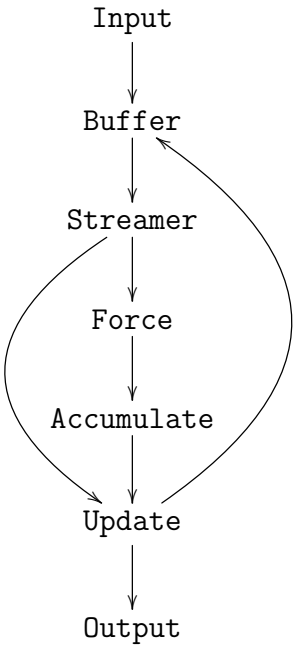


Figure 5.3: `nbody` Topology

n-Body

The next benchmark we consider is the `nbody` benchmark, which 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 given their position, mass, and velocity. The naive algorithm updates each point by considering the

gravitational effect of all other points using the following equation:

$$F_j = G \sum_{i \neq j} \frac{m_j m_i (\vec{p}_j - \vec{p}_i)}{|\vec{p}_j - \vec{p}_i|^3}$$

where G is the gravitational constant, m_i is the mass of the $i^{\texttt{th}}$ particle, and $\vec{p}_i$ is the position of the $i^{\texttt{th}}$ particle.

The topology of the `nbody` benchmark is shown in Figure 5.3. In the `nbody` benchmark, 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 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 benchmark, 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.

Laplace

The `laplace` benchmark is an application to solve Laplace's equation using a Monte-Carlo technique [96]. Laplace's equation is a second-order partial differential equation (PDE) that can be used to model steady-state heat diffusion [108]:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0$$

Given the temperature at the boundaries of an object, solutions to Laplace's equation provide the interior temperatures at equilibrium. Like the n-body simulation, the application to solve Laplace's equation is easily decomposed into a streaming application, as shown in Figure 5.4.

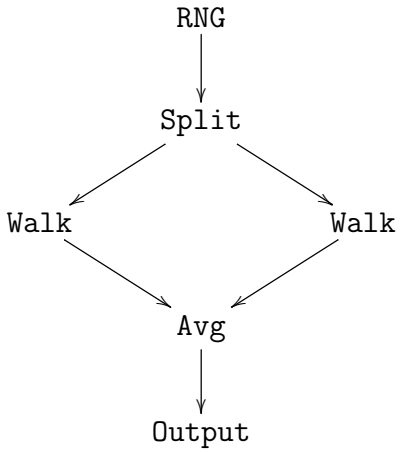


Figure 5.4: `laplace` Topology

In the `laplace` benchmark, random numbers are generated using the Mersenne twister [77] 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 benchmark to use a memory array is the `RNG` kernel, which uses 2,496 bytes of memory. So although this benchmark 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.

Matrix-Matrix Multiply

The `mm` benchmark is a streaming application to perform matrix-matrix multiplication on two 256x256 matrices of 32-bit floats. The topology is shown in Figure 5.5.

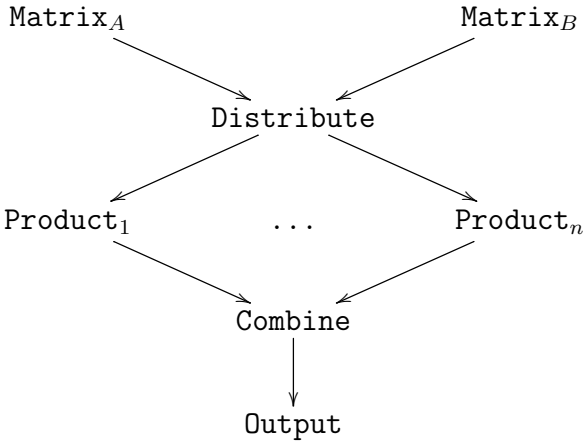


Figure 5.5: `mm` Topology

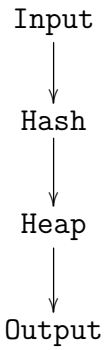


Figure 5.6: `median` Topology

In the `mm` 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: an array to store the matrices totaling 524,288 bytes. In addition to the memory subsystem in the `Distribute` kernel, there are FIFOs connecting all of the kernels, which could potentially be resized.

Median

Finally, we consider the `median` benchmark, which is 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 5.6, 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` benchmark, 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 FIFO between the kernels is another subsystem whose size and implementation is to be optimized.

5.4 Results

As a 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 5.7 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 source of error 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 5.8 shows the actual speedup from running each benchmark on the FPGA device described in Section 5.1. The first bar in each group shows the speedup over the baseline for

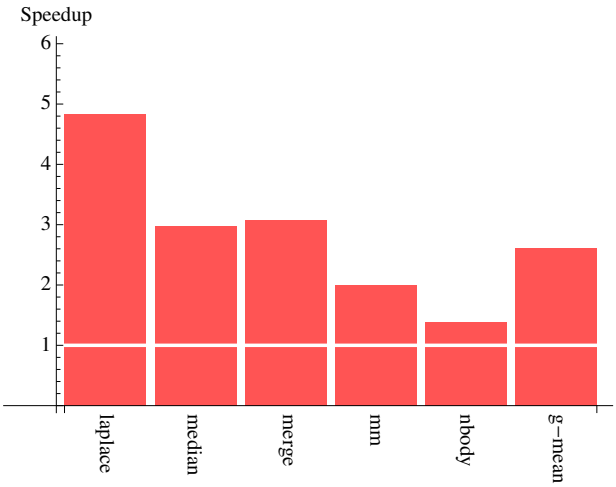


Figure 5.7: Simulated Speedup

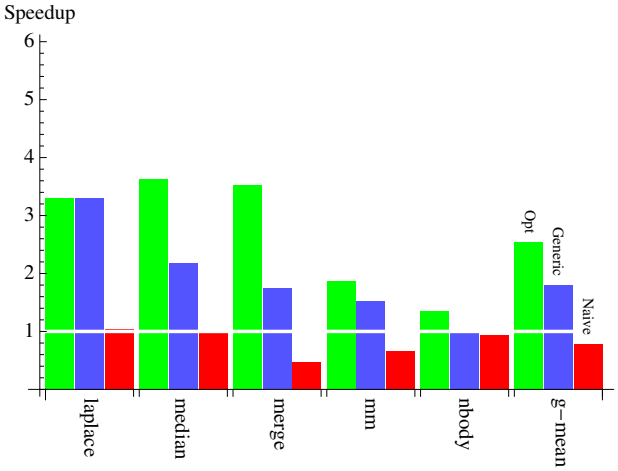


Figure 5.8: Actual Speedup

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. For the *generic* memory subsystem (the second bar in each group in Figure 5.8), 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. The plot shows the speedup of the generic memory subsystem relative to the baseline described earlier.

For the *naive* memory subsystem (the last bar in each group in Figure 5.8), 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. Again, the plot shows the speedup of the naive memory subsystem relative to the baseline.

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.

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 5.2.

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 5.2: `laplace` FIFO Implementations

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 in main memory rather than directly in BRAM. According the 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. Because of this, even the naive memory subsystem offers a performance improvement over the baseline memory subsystem due to the increased FIFO sizes.

Median

For the `median` benchmark, the superoptimized memory subsystem for the `Hash` kernel is shown in Figure 5.9. 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 5.9 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 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 in the case where multiple words are requested within the same 16-byte range.

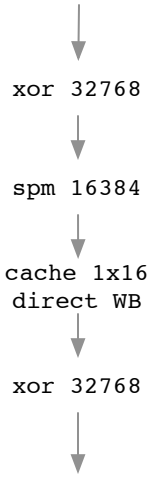


Figure 5.9: Subsystem for the `Hash` Kernel

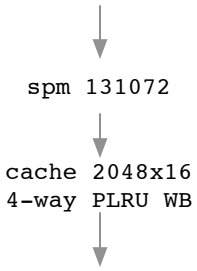


Figure 5.10: Subsystem for the `Heap` Kernel

The superoptimized memory subsystem for the `Heap` kernel is shown in Figure 5.10. 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.

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` 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.

Matrix-Matrix Multiply

As shown in Figure 5.8, 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 memory. The superoptimized memory subsystem for the `Distribute` kernel is shown in Figure 5.11.

There are several interesting features of the memory subsystem shown in Figure 5.11. 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.

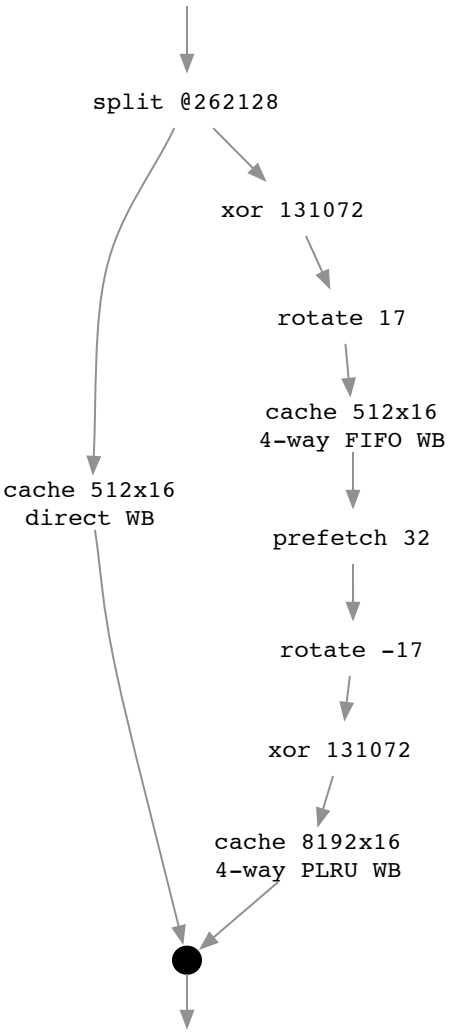


Figure 5.11: Subsystem for the `Distribute` Kernel

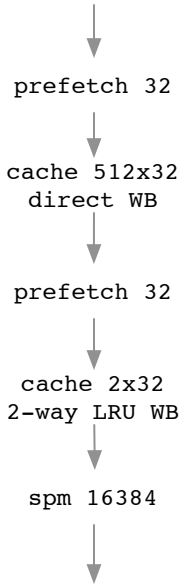


Figure 5.12: Subsystem for the `Buffer` Kernel

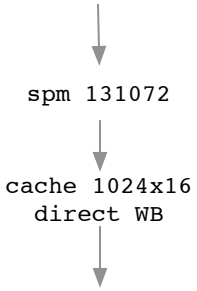


Figure 5.13: Subsystem for the `Streamer` Kernel

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.

n-body

For the `nbody` benchmark, neither the simulated nor actual speedup are very large. This is because the `nbody` 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 5.12. 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 5.13, is a scratchpad followed by a cache. Unlike the previous memory subsystem, in this case the scratchpad is the first part of the memory subsystem. 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 memory subsystem for the `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 do not contribute much to the overall run time. Instead, reducing the memory access times for the `Buffer` kernel provides a greater performance advantage.

5.4.1 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 5.9), which contains an address transformation to move certain parts of the hash table into a scratchpad.

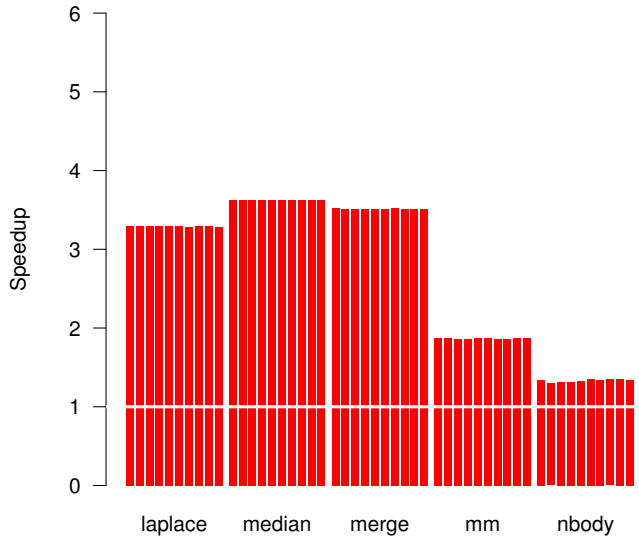


Figure 5.14: Subsystem Specificity

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 5.14. 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.

For the `laplace` benchmark, to change the input we used a different random number seed. As shown in Figure 5.14, 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. As with the `laplace` benchmark, there is little difference in the speedup provided by the superoptimized memory subsystem for both of these benchmarks. Finally, for the `nbody` benchmark, we used a different input size for each run (sized 1,000 to 10,000 in increments of 1,000).

As Figure 5.14 shows, there is very little difference in the performance gain with different input data sets. This implies 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. However, typically this is something we would likely want to avoid.

5.4.2 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 `mm` and `nbody` benchmarks.

The superoptimization process can take a long time. Exactly how long the process takes is dependent 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 simulation time 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. This allows the superoptimizer to revisit prior results without simulation.

Note that the longer the superoptimization process runs, the better the memory subsystem it will discover. However, at all points the memory subsystem is usable. Thus, it is possible to terminate the process as soon as a satisfactory memory subsystem is discovered. For our experiments, the superoptimization process was terminated in an ad-hoc fashion, however, only after a sufficient time such that additional performance gains were infrequent.

5.5 Summary

In this chapter, 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 also 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.

Although the method presented in this chapter to superoptimize streaming applications works well, we note that it quite slow. Therefore, in the next chapter we attempt to improve the time required to superoptimize the memory subsystem for a streaming application through the use of a queuing model.

Chapter 6: A Model for Faster Superoptimization of Streaming Applications

Here we introduce a queuing model for streaming applications to reduce the amount of time required for superoptimizing memory subsystems.

6.1 Introduction

Superoptimization for a single-threaded application is a time-consuming process due to the need to simulate many different memory subsystems. Thus, the superoptimization process for a streaming application consisting of multiple kernels and communication channels can prohibitively time-consuming. This is due to the need to simulate an address trace for all kernels of the application simultaneously where each address trace may be quite long and contain kernel-to-kernel communication along with memory references.

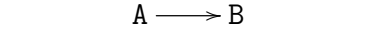


Figure 6.1: Simple Application

To understand how much more computationally intensive the superoptimization process is for a streaming application than for a single-threaded application, we consider the simple streaming application shown in Figure 6.1. This application has two kernels, `A` and `B`, and

a single communication channel. If we assume that the address traces for `A` and `B` are of approximately equal length, this means that the simulation of the system will take twice as long for the streaming application as it would for either kernel individually.

Here we distinguish between memory access events and queue events in kernel traces. A memory access event is a memory read or write whereas a queue event is either a produce or consume. For simplicity, assume that the address traces for both kernels `A` and `B` contain M memory access events and Q queue events. Next, assume that the number of simulations required to find a suitable memory subsystem for kernel `A` is the same as the number of simulations required for `B`, and let that be S. That is, S is the number of simulations that would be required to find a suitable memory subsystem for a kernel if it were treated as a single-threaded application. This means that we can find a suitable memory subsystem for a particular kernel in the streaming application by simulating $S(M + Q)$ events.

For the streaming application, assume that the shared resource constraint does not affect the number of simulations required to find a suitable memory subsystem for a particular kernel. This means that since there are two memory subsystems to discover and two address traces to simulate, the streaming system with two kernels takes $2 \times 2 = 4$ times more event simulations to find a suitable solution than finding a suitable solution for either kernel individually.

In addition to the memory subsystems for the individual kernels, when presented with a streaming application we are also concerned with the memory subsystems for the FIFOs between the kernels. For simplicity, we assume that the only parameter for these FIFOs is their size. This means that we must consider multiple sizes for the FIFOs, which further adds to the number of events that must be simulated.

In general, if we have K kernels in the streaming system, for each proposed memory subsystem we need to simulate $K(M + Q)$ events. If there are F FIFOs and we need to test an average of Z sizes for each proposed memory subsystem, the number of events to simu-

late per proposed memory subsystem increases to $KFZ(M+Q)$. Therefore, the number of events that must be simulated to find a suitable memory subsystem for one of the kernels of the streaming system is $SKFZ(M+Q)$, which is KFZ times more events than required for an individual kernel. Thus, the number of events that must be simulated to find a suitable memory subsystem for all kernels of the streaming system is $SK^2FZ(M+Q)$, which is K^2FZ times more events than for an individual kernel.

Considering how long it takes to find a suitable solution for a individual kernel (about two weeks of total CPU time for the `patricia` benchmark in Chapter 4), this is obviously much longer than ideal even for a modest number of kernels. Therefore, here we describe a method for reducing the number of events that must be simulated to superoptimize the memory subsystem for the streaming application from $SK^2FZ(M+Q) = SK^2FZM + SK^2FZQ$ down to $SKM + SKFZQ$.

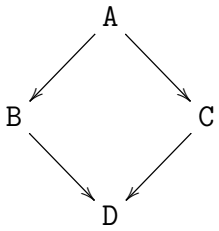


Figure 6.2: Example Topology

The parameters of the queue model are summarized in Table 6.1.

Value	Description
K	The number of kernels in the system
F	The number of FIFOs in the system
M	The mean number of events in a memory address trace
Q	The mean number of events in a queue trace
S	The simulations required to find a suitable memory for a particular kernel
Z	The mean number of FIFO sizes to be tested per queue
t_i	A trace of queue operations for queue i
k_i	Maximum depth of queue i

Table 6.1: Model Parameters

As an example, suppose we have a streaming application with 4 kernels (K) and 4 queues (F), as shown in Figure 6.2. Further, assume there are an average of 100 million memory access instructions (M) and 1 million queue instructions (Q) in the per-kernel traces. These numbers can vary significantly, but are fairly typical. Finally, assume we need 10 thousand iterations (S) to find a suitable memory subsystem and we need to try an average of 5 FIFO sizes (Z) for each queue. Using these figures, we can estimate how many fewer events would need to be simulated using the proposed model.

Using full simulation we get:

$$SK^2FZ(M+Q) = 10^4 \times 4^2 \times 5 \times 5 \times (10^8 + 10^6) \approx 4 \times 10^{14} \text{ events}$$

Using the model:

$$SKM + SKFZQ = 10^4 \times 4 \times 10^8 + 10^4 \times 4 \times 5 \times 5 \times 10^6 \approx 5 \times 10^{12} \text{ events}$$

Thus, using the model we simulate 80 times fewer events, which means a superoptimization process that took months before is now reduced to days.

In addition to the reduction in events to be simulated, our proposed method finds near-optimal queue sizes at all stages rather than leaving the queue sizes as parameters to be optimized. This means that the number of parameters to the superoptimizer is reduced, which translates into a smaller number of required iterations. It should be noted that we still need to simulate the same number of queue events, but queue events are much faster to simulate than memory access events. As a result, the model provides a substantial reduction in time required to superoptimize the memory subsystem for a streaming application.

6.2 Method

To reduce the number of address trace simulations required, we model the streaming system as a queuing network. Then, rather than simulate the memory accesses and queue operations of all kernels, we use a trace of the queue operations for each kernel. A trace of queue operations consists of triples specifying the queue operation (produce or consume), the queue, and the time elapsed since the last queue operation. These traces consist of less data than the full address traces since they do not contain information concerning memory accesses (Q instead of $M+Q$). Further, these traces tend to compress easily (we use LZ77 [137]) and are easy to simulate since simulating a queue is easier than simulating a memory subsystem.

The queue traces are obtained by simulating the full address traces for each kernel in isolation. These full simulations contain queue operations, memory accesses, and computation time. When simulating the full address traces, the simulator assumes that there is always input available for the kernel to consume and the output channels always have space available. This ensures that arrival times and service times are not affected by blocking on the communication channels. The result is a queue trace describing the queue operations that would occur if the kernel were allowed to run without blocking.

Using the queue trace determined from the address trace simulation and holding the maximum queue depths, k_i, constant, we can simulate the queuing network. By "pausing" the queue trace when there is a blocking operation, we can then determine the run time for the kernel in the full application context. The simulation is over once the final queue in the network has exhausted its trace. All but the last queue in the network restart their traces after exhausting them to allow for differences in the number of items consumed due to differing queue depths and split/join kernels. The ending simulation time then provides an approximation of the run time for the full streaming application.

Previously, we left the maximum queue depths as additional parameters to superoptimization. Although we could leave the depths a parameter or even select them a priori, using the queue traces it is easy to determine the optimal queue depths after each kernel trace simulation. To determine the queue depth, k_i, for each queue, we start all queues with depth 1 and simulate the queues. The queue that is blocked most often is the bottleneck. Therefore, the size of that queue is doubled. This process repeats until the performance of the system no longer improves or we run out of resources. Determining the queue depth in this fashion takes longer at each step, but it reduces the number of steps needed for superoptimization while sizing the queues such that they are close to optimal. Doubling the queue sizes rather than incrementing them greatly reduces the number of simulations required, though at the expense of finding the true optimal queue sizes.

In general, a communication channel could be implemented in a number of ways. However, here we assume that the communication channel will be implemented either as a register or as a FIFO implemented in BRAM. From the results in Chapter 5, this seems to be the most common case. Further, this seems to be a logical choice since a BRAM (or register) implementation will always be the fastest and other implementations would contend for main memory bandwidth. Supporting arbitrary memory subsystems for FIFOs would be possible, but would require a more complex queue simulation.

To summarize, once we have queue traces from the kernels, the process to determine the total application run time for a particular memory subsystem is shown in Figure 6.3. The process to determine the best memory subsystem for a streaming application is shown in Figure 6.4.

As can be seen from Figure 6.4, the address trace for only a single kernel needs to be simulated for each modification to a memory subsystem. Thus, each modification, M events need to be simulated to extract an updated queue trace. Next, to simulate the queue network,

```
– Determine run time and queue sizes for queue network n
function GetRunTime(n)
    – Determine initial run time (t) and bottleneck queue (b)
    for all i ∈ Queues do
        k_i ← 1
    end for
    t, b ← SimulateNetwork(n, k)

    – Determine queue sizes
    while ResourcesAvailable do
        – Double the size of the bottleneck queue (b)
        k_b ← 2k_b
        t*, b ← SimulateNetwork(n, k)
        if t* = t then
            – No improvement
            k_b ← k_b/2
            return t, k
        end if
        t ← t*
    end while

    – Out of resources
    return t, k
end function
```

Figure 6.3: Simulation Algorithm

FQ events need to be simulated. The queue network is simulated an average of Z per modification, giving FQZ. This gives a total of $M + FQZ$ events per modification. Since S modifications to the memory subsystem are required per kernel to arrive at a suitable memory subsystem, this means $SM + SFQZ$ total events must be simulated for each kernel, giving $SKM + SKFQZ$ total events for the application.

Of note is that with every modification to a memory subsystem, multiple queuing network simulations must be performed to determine the new queue depths and estimate application run time. Fortunately, these queuing network simulations are typically must faster than the simulation of an address trace. These multiple simulations effectively explore more of the

```
function SUPEROPTIMIZE
    – Initialize memory subsystems (m) and get queue traces (n)
    m ← {∅, ...}
    for all i ∈ Kernels do
        n_i ← SIMULATEKERNEL(m_i, i)
    end for

    – Initialize acceptance threshold (T) and best result (b)
    T, k ← GETRUNTIME(p)
    b_t ← T – Best time
    b_m ← m – Best memory subsystem
    b_k ← k – Best queue sizes

    – Perform optimization
    while TIMEREMAINING do
        – Perturb memory and get new run time
        i ← RANDOM(1, |Kernels|)
        m*_i ← PERTURBSUBSYSTEM(m_i)
        n*_i ← SIMULATEKERNEL(m*_i, i)
        t, k* ← GETRUNTIME(n*)

        – Update best
        if t < b_t then
            b_t ← t – Best time
            b_m ← m
            b_{m,i} ← m*_i – Best memory subsystem
            b_k ← k* – Best queue sizes
        end if

        if t ≤ T then
            – Accept the proposal
            m_i ← m*_i
            n_i ← n*_i
            k ← k*
        end if
        T ← UPDATETHRESHOLD(T, t)
    end while
    return b_m, b_k
end function
```

Figure 6.4: Superoptimization Algorithm

search space than would be explored via the naive algorithm without requiring the simulation of additional memory access events.

6.3 Model Error

Because the queue simulation operates using queue traces, if we assume that the kernels operate independently of each other, there is no error introduced and the result from the queue simulation should match the result from a simulation of the full streaming application. However, the kernels are not completely independent for two reasons. First, there is communication between the kernels over the queues and, second, there is a shared main memory.

As far as the communication between kernels is concerned, we note that this does not actually alter the correctness of the result since the queue simulation models this communication by pausing the traces of blocked kernels. Unfortunately, the shared main memory remains a concern. This is because each of the kernels is run independently to obtain the queue trace, which will have timings from memory accesses independent of other kernels that may contend for memory bandwidth.

Although contention for the shared main memory will likely change the absolute timing results when compared to a full simulation, we do not expect it to affect much the result of superoptimization in most cases. This is because we expect any kernel that is using up significant main memory bandwidth would be a prime candidate for a better memory subsystem. Thus, those memory subsystems will be altered by the superoptimizer to reduce main memory traffic, just as would be the case for a complete simulation. Further, kernels that do not use much main memory bandwidth will likely be affected little by additional delays when they do access main memory. Nevertheless, it is possible that given an application

with multiple kernels making excessive use of main memory bandwidth, the results from the model could vary so much from the results of a full simulation that the superoptimization process would fail to find a suitable memory subsystem.

To avoid the situation where the model and full simulation disagree, the superoptimizer periodically performs a full simulation. If the full simulation disagrees with the model by more than 1%, the superoptimizer will stop using the model and instead use full simulation for its results. This validation ensures that the superoptimizer is able to find a suitable memory subsystem even in cases where the model is inaccurate.

Obviously, a full simulation can be extremely time-consuming. Therefore, validations using full simulation are performed on the initial memory subsystem. Next, the frequency of performing validation using full simulation is backed off in an exponential fashion. If at any point the model is discovered to be inaccurate, full simulation is used for the remainder of the superoptimization process.

6.4 Benchmarks

We use a collection of applications implemented in ScalaPipe [120] for benchmarks. These benchmarks are the same benchmarks as used in Chapter 5. As mentioned in Chapter 3, ScalaPipe is a streaming application generator that allows one to author streaming applications in a high-level language and deploy them either to general-purpose processors or FPGAs. Using ScalaPipe, we can obtain the necessary memory address traces for the kernels automatically. The benchmarks used here are the `laplace`, `median`, `merge`, `mm`, and `nbody` benchmarks.

The topology of the `laplace` benchmark is shown in Figure 5.4. This benchmark finds a solution to Laplace's equation using a Markov-Chain Monte-Carlo technique. For this partic-

ular implementation, only one kernel, `RNG`, uses a memory array. The memory requirement of this kernel is very small, requiring only 2,496 bytes of memory. Therefore, it is reasonable to expect that a superoptimized memory subsystem for this application may not use off-chip memory at all, though additional memory could be used for the FIFOs between the kernels.

Next, the topology of the `median` benchmark is shown in Figure 5.6. This benchmark is a two-stage pipeline to find the median of one million unique 32-bit integers. The first stage (the `Hash` kernel) performs a hash lookup to remove duplicates and the second stage (the `Heap` kernel) performs operations on a binary heap to insert the values and extract the median. Thus, this benchmark has two kernels that require off-chip memory.

The topology of the `merge` benchmark is shown in Figure 5.2. This benchmark sorts a series of one million 32-bit integers using generic merge kernels with a single input channel and a single output channel. Since ScalaPipe uses off-chip memory resources for kernels that use more than 1024 bits of memory, there are 15 kernels in this benchmark that use external memory. Note that there are 20 `Merge` kernels in this benchmark giving a total of 22 kernels and 21 queues.

The topology of the `mm` benchmark is shown in Figure 5.11. The `mm` benchmark is a streaming matrix-matrix multiply implementation. Like the `laplace` benchmark, the `mm` benchmark only has one kernel that accesses external memory, the `Distribute` kernel. However, the memory requirements of the `Distribute` kernel are greater than the `RNG` kernel used in the `laplace` benchmark and off-chip memory is required.

Finally, the topology of the `nbody` benchmark is shown in Figure 5.3. This benchmark performs an n-body simulation using the naive $O(n^2)$ algorithm. Like the `median` benchmark, the `nbody` benchmark has two kernels, `Buffer` and `Streamer`, that access off-chip memory and, therefore, two memory subsystems to be optimized.

6.5 Evaluation

To evaluate our model, we compare the results of superoptimization using the model to those obtained without the model from Chapter 5. We are interested in two objectives. First, we are interested in how the discovered memory subsystems differ. Ideally, there would be little or no difference in the performance of memory subsystems discovered via superoptimization using the model and superoptimization without the model. Second, we are interested in the run time of the superoptimization process. Since the purpose of our model is to reduce this run time, we hope to see a reduction in the amount of run time required to get a similar result.

6.5.1 Subsystem Performance

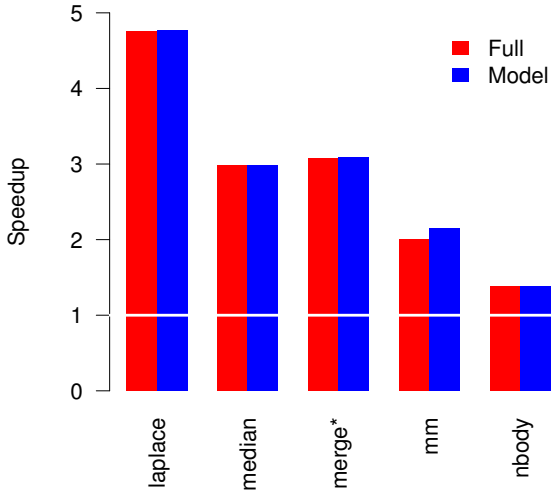


Figure 6.5: Speedup

First we consider how the memory subsystems differ when we use the model for superoptimization from running a full simulation at each step. As in Chapter 5, we use a baseline memory subsystem that uses a single register for all FIFOs between kernels and makes all kernel memory accesses go directly to the main memory. Figure 6.5 compares the speedup of the memory subsystems superoptimized using full simulation (from Chapter 5) against those superoptimized using our model.

Laplace

For the `laplace` benchmark, the performance of the memory subsystem superoptimized using the model is actually slightly better than the performance of the memory subsystem superoptimized using full simulation, however, this difference is very small (less than 1%). Such a situation can arise not only due to the stochastic nature of the search, but also because the model can often do a better job of sizing the FIFOs for each proposed memory subsystem. With the full simulation the FIFO sizes are just another parameter to be explored, and, therefore, it may take many additional simulations to discover the optimal value for the FIFO sizes. As might be expected, the memory subsystem discovered for the `RNG` kernel is the same for the two superoptimization techniques: because the memory footprint of the kernel is so small, a scratchpad suffices to service all accesses.

To understand why there is a difference in performance for the `laplace` benchmark, one must consider the implementation of the FIFOs. Using full simulation, the superoptimizer has more FIFO implementations available. In particular, the superoptimizer has the option to implement the FIFOs in main memory rather than BRAM. Because a main memory implementation uses fewer BRAM resources, the superoptimizer prefers such an implementation. Thus, the memory subsystem discovered using full simulation has several FIFOs implemented in main memory rather than BRAM, as shown in Table 5.2. The memory

FIFO	Model		Full	
	Depth	Implementation	Depth	Implementation
`RNG` → `Split`	1	register	1	register
`Split` → $\texttt{Walk}_1$	2304	BRAM	256	main memory
`Split` → $\texttt{Walk}_2$	1	register	256	BRAM
$\texttt{Walk}_1$ → `Avg`	1152	BRAM	64	main memory
$\texttt{Walk}_2$ → `Avg`	1152	BRAM	8	BRAM
`Avg` → `Output`	1	register	1	register

Table 6.2: Laplace FIFO Comparison

subsystem discovered using the model, on the other hand, has larger FIFOs that are implemented completely in BRAM, whose sizes are shown in Table 6.2. Given enough time, it is likely that the full simulation would arrive at the same solution as the model or possibly better, which uses more resources at the expense of a minor performance improvement.

Median

As with the `laplace` benchmark, for the `median` benchmark the performance of the memory subsystem superoptimized using the model is slightly better than the performance of the memory subsystem superoptimized using full simulation. Superoptimization using the model selected a depth of 13,824 for the FIFO between the `Hash` and `Heap` kernels whereas the superoptimization using the full simulation selected to have a depth of only 16.

For the `median` benchmark, the memory subsystem discovered for the `Heap` kernel using the model, shown in Figure 6.6, is the same as the one previously discovered using full simulation. However, the memory subsystems for the `Hash` kernel differs. The memory subsystem discovered using the model for the `Hash` kernel is shown in Figure 6.8. For comparison, the memory subsystem discovered for the `Hash` kernel using full simulation (from Chapter 5) is shown in Figure 6.7.

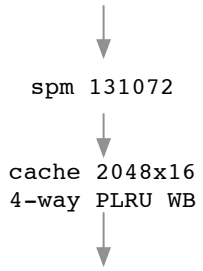


Figure 6.6: Subsystem for the `Heap` Kernel

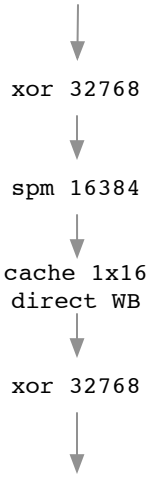


Figure 6.7: Subsystem for the `Hash` Kernel (Full)

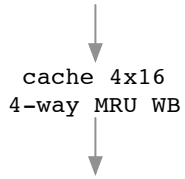


Figure 6.8: Subsystem for the `Hash` Kernel (Model)

Merge Sort

The `merge` benchmark is an interesting case. As previously noted, the `merge` benchmark has 15 kernels contending for main memory access. Thus, it should come as little surprise that the model is not accurate within 1% for this benchmark. In fact, the model differs from full simulation by more than 50%, therefore, the superoptimizer does not use the model for this benchmark. Because the superoptimizer switched to full simulation for this benchmark, the results shown in Figure 6.5 come from superoptimization using full simulation instead of the model.

The question one might ask when confronted with this situation is: would the model still find a good memory subsystem? Unfortunately, in this case it would not. Although the memory subsystem that the superoptimizer discovers for the `merge` benchmark is better than the baseline, the superoptimizer allocates more resources to the FIFOs between the kernels than to the kernels themselves, when, at least in this case, the kernels are the bottleneck due to main memory contention.

Matrix-Matrix Multiply

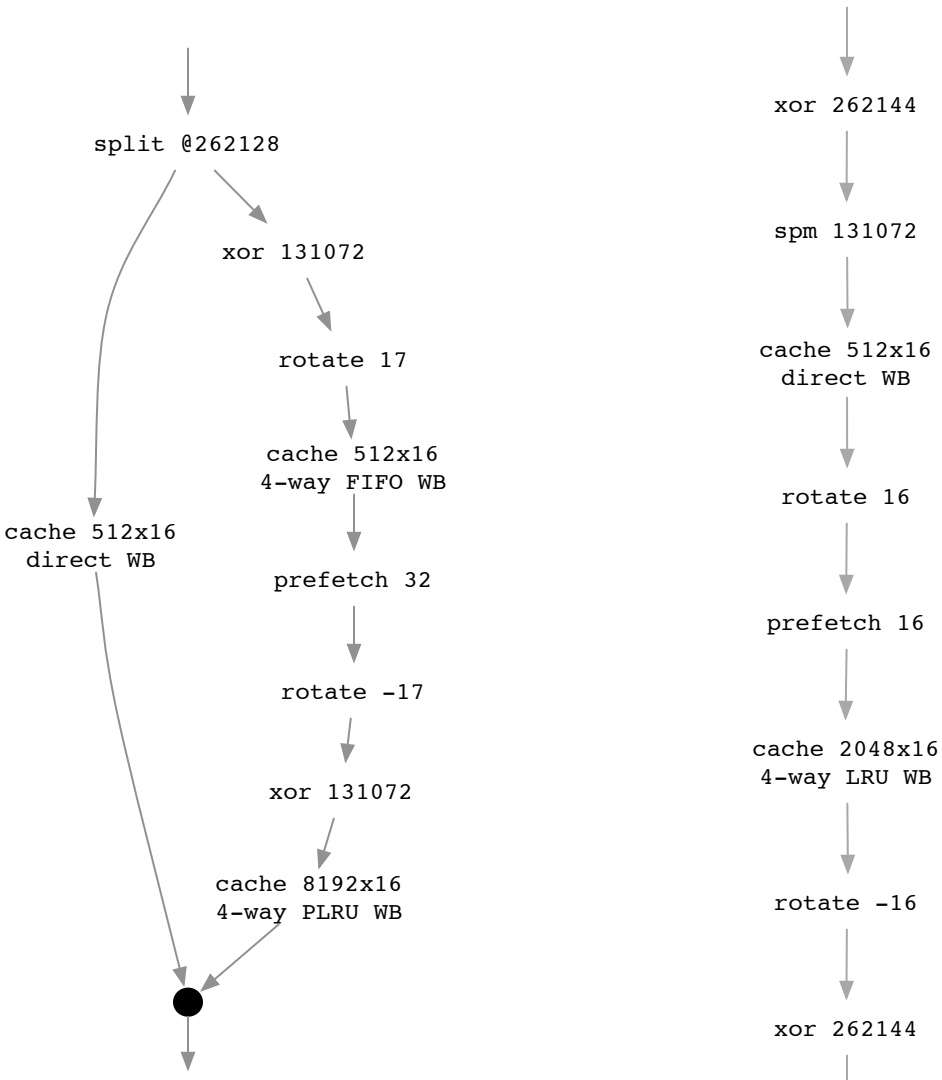
The memory superoptimized for the `Distribute` kernel of the `mm` benchmark using the model is shown in Figure 6.9b. Comparing this memory subsystem to the memory subsystem that was superoptimized using full simulation, shown in Figure 6.9a, we see that while they are different, there are several similar aspects. The differences in the memory subsystems are likely due to the stochastic nature of the search and the fact that, when using the model, the superoptimizer is able to try more distinct memory subsystems since the queue implementations are not search parameters.

In addition to the differences in the memory subsystems themselves, the sizes of the FIFOs differ between the two results. A comparison of the FIFO implementations is shown in Table 6.3. Here we see that the implementation of the FIFOs is the same other than their depths: the FIFOs generated from the model are deeper than those discovered using full simulation.

FIFO	Model		Full	
	Depth	Implementation	Depth	Implementation
$\texttt{Matrix}_A \rightarrow \texttt{Distribute}$	1	register	1	register
$\texttt{Matrix}_B \rightarrow \texttt{Distribute}$	1	register	1	register
$\texttt{Distribute} \rightarrow \texttt{Product}_1$	1152	BRAM	256	BRAM
$\texttt{Distribute} \rightarrow \texttt{Product}_2$	1152	BRAM	256	BRAM
$\texttt{Product}_1 \rightarrow \texttt{Combine}$	1152	BRAM	128	BRAM
$\texttt{Product}_2 \rightarrow \texttt{Combine}$	1152	BRAM	128	BRAM
$\texttt{Combine} \rightarrow \texttt{Output}$	1	register	1	register

Table 6.3: Matrix-Matrix Multiply FIFO Comparison

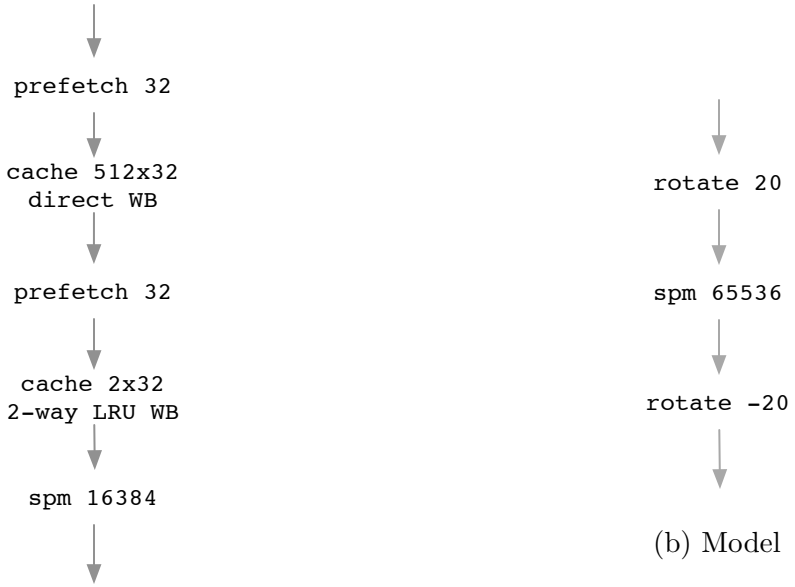
From Figure 6.5, we see that the memory subsystem superoptimized using the model is able to provide a greater speedup than the memory subsystem superoptimized using full simulation. There are two possible reasons for this. First, as previously noted, the memory subsystems are different. Secondly, the FIFO depths are different. In this case, although the



(a) Full

(b) Model

Figure 6.9: Subsystems for the `Distribute` Kernel



(a) Full

(b) Model

Figure 6.10: Subsystems for the `Buffer` Kernel

deeper FIFOs explain a small part of the performance difference, most of the difference is explained by the memory subsystem.

n-body

The memory subsystem superoptimized using the model for the `Buffer` kernel is shown in Figure 6.10b. Compared to the memory subsystem superoptimized using full simulation, shown in Figure 6.10a, the memory subsystem superoptimized using the model is much simpler. Likewise, the memory subsystems for the `Streamer` kernel are shown in Figure 6.11b (model) and Figure 6.11a (full simulation). Again, the memory subsystem superoptimized using the model is simpler.

Comparing the performance of the memory subsystems in Figure 6.5, we see that there is little difference in performance between the two memory subsystems. However, from the raw data we see that the memory subsystem superoptimized using full simulation performs

(a) Full

(b) Model

Figure 6.11: Subsystems for the `Streamer` Kernel

slightly better than the memory subsystem superoptimized using the model (though there is less than a 1% difference). This is likely due to a discrepancy in the performance reported from the model and full simulation. The model shows that the memory subsystem discovered using the model performs better even though the other memory subsystem performs better in reality. Because the memory subsystems are so similar in performance, however, the superoptimizer does not switch to full simulation and we are left with a memory subsystem that performs slightly worse than we would likely have had we used full simulation.

6.5.2 Superoptimization Run Time

Finally, we consider the amount of time required to superoptimize the memory subsystem for a streaming application. Recall that the model presented here was able to correctly predict the performance (within 1%) of 4 of the 5 benchmarks; full simulation was required for the `merge` benchmark. Here we use the `median` benchmark to compare superoptimization using the model with superoptimization using full simulation.

On our test system, a full simulation of memory trace for the `median` benchmark with an empty memory subsystem takes about 90 seconds. In isolation, the `Hash` kernel takes about 4 seconds to simulate and the `Median` kernel takes about 40 seconds, giving an average of

22 seconds per kernel containing a memory subsystem. However, to use our queue model we need queue traces rather than a timing result, which makes the simulation take slightly longer: 5 seconds for the `Hash` kernel and 51 seconds for the `Median` kernel, for an average of 33 seconds.

To get the equivalent of a full simulation using our model, we need to simulate one of the memory subsystems, which takes an average of 33 seconds, and then perform a simulation of the queuing network for all queue sizes of interest. The queue sizes are doubled for the bottleneck queue, which makes the maximum number of queuing network simulations required logarithmic in the number of resources remaining after allocating resources to the memory subsystems.

For our experimental platform, we are given 92 BRAMs. Each BRAM can support up to 512 4-byte entries in a queue. If we assume that the bottleneck does not move around, this means that we are limited to $\lceil \lg 512 \times 92 \rceil = 16$ simulations per step. The situation is slightly worse if we assume that the bottleneck moves around after each simulation, since we must then multiply by the number of queues: $16 \times 3 = 48$ simulations. Thus, in the worse case, one step using the model could take $33 + 48 \times 3 = 177$ seconds. However, typically the bottleneck does not move around and we usually have fewer than 92 BRAMs remaining (since the memory subsystems use them). Assuming a typical case where the bottleneck does not move around and we have half of the BRAMs available, this means one step takes $33 + \lceil \lg 512 \times 92/2 \rceil \times 3 = 78$ seconds.

Although it may seem a reduction from 90 seconds per iteration (using full simulation) to 78 seconds (using the model) is insignificant, it is important to note that each iteration provides much more information when using the model. In particular, since the sizes of the FIFOs are determined during the simulation of the queue network, so in practice many fewer iterations are required. How many fewer iterations are required is a function of the number of resources

available and the number of queues. For the `median` benchmark, if we assume half of the 92 BRAM resources can be allocated to queues, given that there are three queues of interest, this is a reduction of somewhere near $(92/2) \times 3 = 138$ times. However, the superoptimizer may explore only a fraction of these when using full simulation.

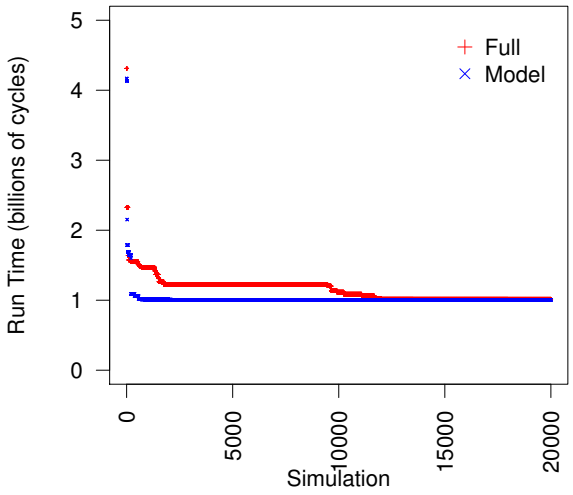


Figure 6.12: Simulations Required for Superoptimization

Figure 6.12 compares the best result after each simulation for the `median` benchmark using both superoptimization with full simulation and superoptimization with the model. To make the results more comparable, for this experiment the superoptimization process assumes all FIFOs are implemented in BRAM rather than using the implementation as yet another parameter, as was done in Chapter 5. Here we see that not only does the model speed up the time required for each simulation, but, because the queue simulation determines near-optimal queue sizes after each simulation, the model allows the superoptimization to find good results with fewer simulations. Note that the superoptimization process with full simulation leaves the queue sizes as another superoptimization parameter. Thus, when the model works it provides results much more quickly than would otherwise be possible.

6.6 Summary

In this chapter we presented a model to reduce the amount of time required to superoptimize the memory subsystem for a streaming application. This model allows one to superoptimize memory subsystems more quickly and approach problems where it might not otherwise be reasonable to use superoptimization. We showed that although this model makes a simplifying assumption about main memory bandwidth, it still allows us to discover memory subsystems that are similar to those discovered using full simulation and sometimes better due to the improved method of finding queue sizes. There are cases where where the model breaks down when there is excessive main memory contention, but we are able to detect this easily. For most of our benchmarks we were able to use the model to realize significant reductions in the time to perform superoptimization.

Chapter 7: Conclusion

In this dissertation, we investigated the use of highly-specialized memory subsystems for applications. In particular, we investigated combinations of caches, scratchpads, address transformations, split address spaces, and other components in the memory subsystem. Combining these components, we showed that it is possible to produce memory subsystems that out-perform traditional memory subsystems, such as cache hierarchies. To do this, we provided a superoptimization technique to discover custom memory subsystems for single-threaded applications to be deployed on FPGAs and ASICs. This required the development of tools including a memory subsystem simulator and a memory subsystem superoptimizer.

Although performance is our primary focus in this work, we also showed that our superoptimization technique is generic by optimizing to reduce writes to main memory. Reducing writes to main memory is an increasingly important objective since an increasing number of alternative main memory technologies, such as Phase-Change Memory (PCM) and Flash, have an aversion to writes. This aversion stems from limited write endurance, increased energy from writes, and long write latencies. Our results show that there is gain to be had by taking writes into account.

Next, we investigated the superoptimization of memory subsystems for streaming applications, which are a class of parallel applications. To this end, we extended our superoptimization technique to support streaming applications. To obtain traces for the parallel applications, we used applications developed in ScalaPipe, which is another tool we developed.

Our results revealed that impressive performance improvements are possible with streaming applications, both in simulation and when deployed on an FPGA device. Unfortunately, these results take a long time to obtain due to the lengthy search process required.

Finally, to address the long search process required for the superoptimization of streaming applications, we developed a model to allow us to reduce the number of events that need to be simulated. Using this model, we are able to significantly reduce the run time of the superoptimization process for many applications. Although the model breaks down in some cases, it appears to work well in most case and we are able to identify those cases where the model does not work.

7.1 Future Work

There are many possible directions for future work. Here we describe several possibilities.

Other Memory Components An obvious extension of this work is the consideration of other memory subsystem components. Unfortunately, adding additional components would likely make the superoptimization process more time-consuming. Therefore, finding the right mix of components to be able to obtain good results in a reasonable amount of time would be desirable.

Datapath Optimization Although this work considers only memory subsystem optimization and not data path optimization or topology optimization, optimizing these simultaneously could lead to better results. For example, it is often the case that the number

of kernels can be increased to increase parallelism at the expense of more resources and additional memory bandwidth contention.

Faster Superoptimization Despite the heuristics presented here and the queuing model used for streaming applications, the superoptimization process is still extremely time consuming. There remain several techniques that could be investigated to improve the situation. For example, using some of the previous work on speeding up cache simulations could be incorporated into the superoptimizer to allow it to evaluate multiple caches simultaneously. Also, it may be possible to classify application behavior using a model rather than precisely with an address trace.

Improved Model Although the queuing model described in Chapter 6 works in many cases, it can become inaccurate when there is excessive main memory contention. In this work we simply validate the model periodically and switch to the slower method of using full simulation when the model falls apart. A better solution, however, would be to modify the model to account for main memory contention.

Application Phases In this work we assumed that each application or kernel could only use a single memory subsystem throughout its execution. However, many applications exhibit distinct phases of execution such that it could be useful to alter the memory subsystem at run time [102]. Supporting multiple superoptimized memory subsystems for different application phases represents an interesting opportunity for future work.

Multiple Application Support Although we have investigated superoptimized memory subsystems for single-threaded and parallel-applications, we have not considered any form of resource sharing among multiple applications. Extending this work to support virtual mem-

ory and/or multiple running applications represents an interesting opportunity for future work. Because our superoptimized memory subsystems are application-specific, changing the subsystem for a particular application would likely have a significant overhead that would need to be considered in the superoptimization process. Further, the issue of virtual memory presents unique challenges for simulation since it is usually not possible to determine ahead of time what physical addresses will be used.

Other Models of Parallelism Here we described the streaming paradigm for parallelism due to the explicit nature of the communication channels and independent memory subsystems for kernels. However, shared-memory parallelism is extremely common today. To work toward a completely shared-memory type of parallelism, it may be possible to support specific types of shared data structures, such as queues, hashes, and locks.

Better General-Purpose Memories Our focus has been on application-specific memory subsystems, but the technique proposed here could be used for general-purpose memories. Using superoptimization for general-purpose memories could find novel memory subsystems that perform better than traditional cache hierarchies.

References

[1] Michael Adler, Kermin E Fleming, Angshuman Parashar, Michael Pellauer, and Joel Emer. LEAP scratchpads: automatic memory and cache management for reconfigurable logic. In *Proc. of 19th ACM/SIGDA Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, pages 25–28, February 2011.

[2] Hussein Al-Zoubi, Aleksandar Milenkovic, and Milena Milenkovic. Performance evaluation of cache replacement policies for the SPEC CPU2000 benchmark suite. In *Proc. of 42nd Southeast Regional Conference*, pages 267–272, 2004.

[3] ARM1136JF-S and ARM1136J-S technical reference manual. Technical Report 0211K, ARM Holdings plc, February 2009.

[4] Rajeev Balasubramonian, David H Albonesi, Alper Buyuktosunoglu, and Sandhya Dwarkadas. A dynamically tunable memory hierarchy. *IEEE Trans. on Computers*, 52(10):1243–1258, October 2003.

[5] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, M Balakrishnan, and Peter 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.

[6] Sorav Bansal and Alex Aiken. Automatic generation of peephole superoptimizers. In *ACM SIGPLAN Notices*, volume 41, pages 394–403. ACM, 2006.

[7] Sorav Bansal and Alex Aiken. Binary translation using peephole superoptimizers. In *Proc. of 8th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, volume 8, pages 177–192, December 2008.

[8] Michela Becchi, Mark Franklin, and Patrick Crowley. Performance/area efficiency in embedded chip multiprocessors with micro-caches. In *Proc. of 4th ACM Intl Conf. on Computing Frontiers*. ACM, May 2007.

[9] Luca Benini, Alberto Macii, and Massimo Poncino. Energy-aware design of embedded memories: A survey of technologies, architectures, and optimization techniques. *ACM Transactions on Embedded Computing Systems (TECS)*, 2(1):5–32, 2003.

[10] Kristof Beyls and Erik H D'Hollander. Refactoring for data locality. *Computer*, 42(2):62–71, 2009.

[11] Roberto Bez, Emilio Camerlenghi, Alberto Modelli, and Angelo Visconti. Introduction to flash memory. *Proceedings of the IEEE*, 91(4):489–502, 2003.

[12] Shekhar Borkar and Andrew A Chien. The future of microprocessors. *Communications of the ACM*, 54(5):67–77, 2011.

[13] Martin Brain, Tom Crick, Marina De Vos, and John Fitch. TOAST: Applying answer set programming to superoptimisation. *Logic Programming*, page 270, 2006.

[14] Brad Calder, Chandra Krintz, Simmi John, and Todd Austin. Cache-conscious data placement. In *Proc. of 8th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 139–149, 1998.

[15] Steve Carr, Kathryn S. McKinley, and Chau-Wen Tseng. Compiler optimizations for improving data locality. In *Proc. of 6th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 252–262, 1994.

[16] Hassan Chafi, Zach DeVito, Adriaan Moors, Tiark Rompf, Arvind K. Sujeeth, Pat Hanrahan, Martin Odersky, and Kunle Olukotun. Language virtualization for heterogeneous parallel computing. In *Proc. of ACM Int'l Conf. on Object Oriented Programming Systems, Languages, and Applications*, pages 835–847, 2010.

[17] Roger D. Chamberlain, Mark A. Franklin, Eric J. Tyson, James H. Buckley, Jeremy Buhler, Greg Galloway, Saurabh Gayen, Michael Hall, E.F. Berkley Shands, and Naveen Singla. Auto-Pipe: Streaming applications on architecturally diverse systems. *Computer*, 43(3):42–49, March 2010.

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

[19] Roger D Chamberlain, Joseph M Lancaster, and Ron K Cytron. Visions for application development on hybrid computing systems. *Parallel Computing*, 34(4):201–216, 2008.

[20] Jichuan Chang, Parthasarathy Ranganathan, David Andrew Roberts, Mehul A Shah, and John Sontag. Data storage apparatus and methods, March 2012. US Patent App. 2012/0131278.

[21] Yuan-Hao Chang, Jian-Hong Lin, Jen-Wei Hsieh, and Tei-Wei Kuo. A strategy to emulate NOR flash with NAND flash. *ACM Transactions on Storage (TOS)*, 6(2):5, 2010.

[22] Mainak Chaudhuri. Pseudo-LIFO: the foundation of a new family of replacement policies for last-level caches. In *Proc. of 42nd IEEE/ACM Int'l Symp. on Microarchitecture*, pages 401–412, 2009.

[23] Yu-Ting Chen, Jason Cong, and Glenn Reinman. HC-Sim: a fast and exact L1 cache simulator with scratchpad memory co-simulation support. In *Proc. of 9th Int'l Conf. on Hardware/Software Codesign and System Synthesis (CODES+ ISSS)*, pages 295–304. IEEE, 2011.

[24] Trishul M. Chilimbi, Bob Davidson, and James R. Larus. Cache-conscious structure definition. In *Proc. of ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 13–24, 1999.

[25] Trishul M Chilimbi, Mark D Hill, and James R Larus. Cache-conscious structure layout. In *Proc. of ACM Conf. on Programming Language Design and Implementation*, pages 1–12, 1999.

[26] Young-kyu Choi, Jason Cong, and Di Wu. FPGA implementation of EM algorithm for 3D CT reconstruction. In *Proc. of 22nd Symp. on Field-Programmable Custom Computing Machines (FCCM)*, pages 157–160. IEEE, 2014.

[27] Eric S Chung, James C Hoe, and Ken Mai. CoRAM: an in-fabric memory architecture for FPGA-based computing. In *Proc. of 19th ACM/SIGDA Int'l Symp. on Field-Programmable Gate Arrays*, pages 97–106, February 2011.

[28] Jason Cong, Muhuan Huang, and Peng Zhang. Combining computation and communication optimizations in system synthesis for streaming applications. In *Proc. of 22nd Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, pages 213–222. ACM, 2014.

[29] Charles Consel, Hedi Hamdi, Laurent Réveillère, Lenin Singaravelu, Haiyan Yu, and Calton Pu. Spidle: a DSL approach to specifying streaming applications. In *Generative Programming and Component Engineering*, pages 1–17. Springer, 2003.

[30] Pat Conway, Nathan Kalyanasundharam, Gregg Donley, Kevin Lepak, and Bill Hughes. Cache hierarchy and memory subsystem of the AMD Opteron processor. *IEEE Micro*, 30(2):16–29, 2010.

[31] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions Programming Languages and Systems (TOPLAS)*, 13(4):451–490, October 1991.

[32] Jack J Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain S Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software (TOMS)*, 16(1):1–17, 1990.

[33] Jack J Dongarra, Piotr Luszczek, and Antoine Petitet. The LINPACK benchmark: past, present and future. *Concurrency and Computation: Practice and Experience*, 15(9):803–820, August 2003.

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

[35] James P Durbano and Fernando E Ortiz. FPGA-based acceleration of the 3D finite-difference time-domain method. In *Proc. of 12th Symp. on Field-Programmable Custom Computing Machines (FCCM)*, pages 156–163. IEEE, 2004.

[36] Matteo Frigo, Charles E Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *Proc. of 40th Symp. on Foundations of Computer Science*, pages 285–297, 1999.

[37] Kyle Gallivan, William Jalby, Ulrike Meier, and Ahmed H Sameh. Impact of hierarchical memory systems on linear algebra algorithm design. *International Journal of High Performance Computing Applications*, 2(1):12–48, 1988.

[38] Stuart Geman and Donald Geman. Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, (6):721–741, 1984.

[39] Arijit Ghosh and Tony Givargis. Analytical design space exploration of caches for embedded systems. In *Design, Automation and Test in Europe Conference and Exhibition*, pages 650–655. IEEE, 2003.

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

[41] Ann Gordon-Ross, Frank Vahid, and Nikil 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.

[42] Ann Gordon-Ross, Frank Vahid, and Nikil 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.

[43] Kazushige Goto and Robert A Geijn. Anatomy of high-performance matrix multiplication. *ACM Transactions on Mathematical Software (TOMS)*, 34(3):12, 2008.

[44] Torbjörn Granlund and Richard Kenner. Eliminating branches using a superoptimizer and the GNU C compiler. In *Proc. of ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 341–352, 1992.

[45] Clemens Grelck, Sven-Bodo Scholz, and Alex Shafarenko. A gentle introduction to S-Net: Typed stream processing and declarative coordination of asynchronous components. *Parallel Processing Letters*, 18(2):221–237, 2008.

[46] Jayanth Gummaraju, Joel Coburn, Yoshio Turner, and Mendel Rosenblum. Streamware: programming general-purpose multicore processors using streams. In *Proc. of 13th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 297–307. ACM, March 2008.

[47] Matthew R Guthaus, Jeffrey S Ringenberg, Dan Ernst, Todd M Austin, Trevor Mudge, and Richard B Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proc. of 4th Int'l Workshop on Workload Characterization*, pages 3–14, 2001.

[48] Mohammad Shihabul Haque, Jorgen Peddersen, Andhi Janapsatya, and Sri Parameswaran. Dew: A fast level 1 cache simulation approach for embedded processors with FIFO replacement policy. In *Proc. of Conf. on Design, Automation and Test in Europe*, pages 496–501. European Design and Automation Association, 2010.

[49] Eli Harari. Flash memory-the great disruptor! In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 10–15. IEEE, 2012.

[50] Charles AR Hoare. Quicksort. *The Computer Journal*, 5(1):10–16, 1962.

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

[52] Intel XScale® core developer's manual. Technical Report 273473-002, Intel Corporation, January 2004.

[53] Intel® 64 and IA-32 architectures optimization reference manual. Technical Report 248966-029, Intel Corporation, March 2014.

[54] Engin İpek, Sally A McKee, Rich Caruana, Bronis R de Supinski, and Martin Schulz. Efficiently exploring architectural design spaces via predictive modeling. In *Proc. of 12th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 195–206, 2006.

[55] Bruce Jacob, Spencer Ng, and David Wang. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann, 2010.

[56] Andhi Janapsatya, Aleksandar Ignjatovic, and Sri Parameswaran. Finding optimal L1 cache configuration for embedded systems. In *Proc. of Asia and South Pacific Conference on Design Automation*, pages 796–801. IEEE, 2006.

[57] Song Jiang and Xiaodong Zhang. LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. *ACM SIGMETRICS Performance Evaluation Review*, 30(1):31–42, 2002.

[58] Lizy Kurian John and Akila Subramanian. Design and performance evaluation of a cache assist to implement selective caching. In *Proc. of Int'l Conf. on Computer Design (ICCD)*, pages 510–518. IEEE, 1997.

[59] Rajeev Joshi, Greg Nelson, and Keith Randall. Denali: a goal-directed superoptimizer. In *Proc. of ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 304–314. ACM, June 2002.

[60] Norman 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.

[61] Martin Kampe, Per Stenstrom, and Michel Dubois. Self-correcting LRU replacement policies. In *Proc. of 1st Conf. on Computing Frontiers*, pages 181–191, 2004.

[62] Johnson Kin, Munish Gupta, and William H Mangione-Smith. The filter cache: an energy efficient memory structure. In *Proc. of 30th ACM/IEEE Int'l Symp. on Microarchitecture*, pages 184–193. IEEE, 1997.

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

[64] Emre Kultursay, Mahmut Kandemir, Anand Sivasubramaniam, and Onur Mutlu. Evaluating STT-RAM as an energy-efficient main memory alternative. In *Proc. of IEEE Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, pages 256–267. IEEE, 2013.

[65] Kanishka Lahiri, Anand Raghunathan, and Sujit Dey. Design space exploration for optimizing on-chip communication architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23(6):952–961, 2004.

[66] Joseph M. Lancaster, E. F. Berkley Shands, Jeremy D. Buhler, and Roger D. Chamberlain. Timetrial: A low-impact performance profiler for streaming data applications. In *Proc. of IEEE Int'l Conf. on Application-Specific Systems, Architectures, and Processors (ASAP)*, pages 69–76. IEEE, September 2011.

[67] Alvin R Lebeck and David A Wood. Cache profiling and the SPEC benchmarks: A case study. *Computer*, 27(10):15–26, 1994.

[68] Benjamin C Lee and David M Brooks. Accurate and efficient regression modeling for microarchitectural performance and power prediction. In *Proc. of 12th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 185–194, 2006.

[69] Benjamin C Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable DRAM alternative. *ACM SIGARCH Computer Architecture News*, 37(3):2–13, 2009.

[70] Dennis C Lee, Patrick J Crowley, Jean-Loup Baer, Thomas E Anderson, and Brian N Bershad. Execution characteristics of desktop applications on Windows NT. *ACM SIGARCH Computer Architecture News*, 26(3):27–38, 1998.

[71] Martin Lukasiewycz, Michael Glaß, Christian Haubelt, and Jürgen Teich. Efficient symbolic multi-objective design space exploration. In *Proc. of Asia and South Pacific Design Automation Conference*, pages 691–696. IEEE Computer Society Press, 2008.

[72] John MacCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, 3(4):184–195, 1960.

[73] Doug MacGregor, Dave Mothersole, and Bill Moyer. The Motorola MC68020. *IEEE Micro*, 4(4):101–118, 1984.

[74] Jack A Mandelman, Robert H Dennard, Gary B Bronner, John K DeBrosse, Rama Divakaruni, Yujun Li, and Carl J Radens. Challenges and future directions for the scaling of dynamic random-access memory (DRAM). *IBM Journal of Research and Development*, 46(2.3):187–212, 2002.

[75] R Timothy Marler and Jasbir S Arora. Survey of multi-objective optimization methods for engineering. *Structural and Multidisciplinary Optimization*, 26(6):369–395, 2004.

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

[77] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation*, 8(1):3–30, January 1998.

[78] David May, Dan Page, James Irwin, and Henk L Muller. Microcaches. In *Proc. of 6th Int'l Conf. on High Performance Computing (HiPC)*, pages 21–27. Springer, December 1999.

[79] Ann Marie Grizzaffi Maynard, Colette M Donnelly, and Bret R Olszewski. Contrasting characteristics and cache performance of technical and multi-user commercial workloads. In *ACM SIGPLAN Notices*, volume 29, pages 145–156. ACM, 1994.

[80] Sally A McKee. Reflections on the memory wall. In *Proc. of 1st Conf. on Computing Frontiers*, page 162, 2004.

[81] Sally A McKee and William A Wulf. Access ordering and memory-conscious cache utilization. In *Proc. of 1st IEEE Symp. on High-Performance Computer Architecture (HPCA)*, pages 253–262. IEEE, 1995.

[82] Matteo Monchiero, Ramon Canal, and Antonio González. Design space exploration for multicore architectures: a power/performance/thermal view. In *Proc. of 20th Int'l Conf. on Supercomputing*, pages 177–186. ACM, 2006.

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

[84] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proc. of ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 89–100, 2007.

[85] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An overview of the Scala programming language. Technical Report IC/2004/64, École Polytechnique Fédérale de Lausanne, 2004.

[86] Shobana Padmanabhan, Yixin Chen, and Roger D. Chamberlain. Convexity in non-convex optimizations of streaming applications. In *Proc. of IEEE 18th Int'l Conf. on Parallel and Distributed Systems (ICPADS)*, pages 668–675, December 2012.

[87] Gianluca Palermo, Cristina Silvano, and Vittorio Zaccaria. Discrete particle swarm optimization for multi-objective design space exploration. In *Proc. of 11th Conf. on Digital System Design Architectures, Methods and Tools*, pages 641–644. IEEE, 2008.

[88] Maurizio Palesi and Tony Givargis. Multi-objective design space exploration using genetic algorithms. In *Proc. of 10th Int'l Symp on Hardware/Software Codesign (CODES)*, pages 67–72. IEEE, 2002.

[89] 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.

[90] Jeff Parkhurst, John Darringer, and Bill Grundmann. From single core to multi-core: preparing for a new exponential. In *Proc. of IEEE/ACM Int'l Conf. on Computer-Aided Design*, pages 67–72. ACM, 2006.

[91] `http://www.postgresql.org`.

[92] Moinuddin K Qureshi, Aamer Jaleel, Yale N Patt, Simon C Steely, and Joel Emer. Adaptive insertion policies for high performance caching. In *ACM SIGARCH Computer Architecture News*, volume 35, pages 381–391, 2007.

[93] Moinuddin K Qureshi, Vijayalakshmi Srinivasan, and Jude A Rivers. Scalable high performance main memory system using phase-change memory technology. *ACM SIGARCH Computer Architecture News*, 37(3):24–33, 2009.

[94] P Ranjan Panda, Nikil D Dutt, Alexandru Nicolau, Francky Catthoor, Amout Vandecappelle, Erik Brockmeyer, Chidamber Kulkarni, and Eddy De Greef. Data memory organization and optimizations in application-specific systems. *IEEE Design & Test of Computers*, 18(3):56–68, 2001.

[95] `http://www.raspberrypi.org`.

[96] JF Reynolds. A proof of the random-walk method for solving Laplace's equation in 2-D. *The Mathematical Gazette*, pages 416–420, 1965.

[97] Tiark Rompf and Martin Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. In *Proc. of 9th Int'l Conf. on Generative Programming and Component Engineering*, pages 127–136, 2010.

[98] John P Scheible. A survey of storage options. *Computer*, 35(12):42–46, 2002.

[99] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. In *Proc. of 18th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 305–316, March 2013.

[100] Sandeep Sen, Siddhartha Chatterjee, and Neeraj Dumir. Towards a theory of cache-efficient algorithms. *Journal of the ACM*, 49(6):828–858, November 2002.

[101] Muhammad Shafiq, Miquel Pericas, Raul de la Cruz, Mauricio Araya-Polo, Nacho Navarro, and Eduard Ayguadé. Exploiting memory customization in FPGA for 3D stencil computations. In *Proc. of Int'l Conf. on Field-Programmable Technology (FPT)*, pages 38–45. IEEE, 2009.

[102] Xipeng Shen, Yutao Zhong, and Chen Ding. Locality phase prediction. In *Proc. of 11th Int'l Conf. on Architecture Support for Programming Languages and Operating Systems (ASPLOS)*, volume 39, pages 165–176, November 2004.

[103] Alan Jay Smith. Cache memories. *ACM Computing Surveys (CSUR)*, 14(3):473–530, 1982.

[104] Byoungro So, Mary W Hall, and Pedro C Diniz. A compiler approach to fast hardware design space exploration in FPGA-based systems. In *ACM SIGPLAN Notices*, volume 37, pages 165–176. ACM, 2002.

[105] Jesper H Spring, Jean Privat, Rachid Guerraoui, and Jan Vitek. StreamFlex: high-throughput stream programming in Java. *ACM SIGPLAN Notices*, 42(10):211–228, 2007.

[106] Vinoo Srinivasan, Shankar Radhakrishnan, and Ranga Vemuri. Hardware software partitioning with integrated hardware design space exploration. In *Proc. of Design, Automation and Test in Europe*, pages 28–35. IEEE, 1998.

[107] John E. Stone, David Gohara, and Guochun Shi. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in Science and Engineering*, 12:66–73, 2010.

[108] Walter A Strauss. *Partial Differential Equations: An Introduction*. Wiley, 1992.

[109] Jeffrey Stuecheli, Dimitris Kaseridis, Hillery C Hunter, and Lizy K John. Elastic refresh: Techniques to mitigate refresh penalties in high density memory. In *Proc. of 43rd IEEE/ACM Int'l Symp. on Microarchitecture*, pages 375–384, 2010.

[110] Ching-Long Su and Alvin M Despain. Cache design trade-offs for power and performance optimization: a case study. In *Proc. of Int'l Symp. on Low Power Design*, pages 63–68. ACM, 1995.

[111] Karthik T Sundararajan, Timothy M Jones, and Nigel 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.

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

[113] Shyamkumar Thoziyoor, Naveen Muralimanohar, Jung Ho Ahn, and Norman P Jouppi. CACTI 5.1. *HP Laboratories*, 2, April 2008.

[114] Sumesh Udayakumaran. *Compiler-Decided Dynamic Memory Allocation for Scratch-Pad Based Embedded Systems*. PhD thesis, Univ. of Maryland, 2006.

[115] Steven P Vanderwiel and David J Lilja. Data prefetch mechanisms. *ACM Computing Surveys (CSUR)*, 32(2):174–199, 2000.

[116] Jasmina Vasiljevic and Paul Chow. MPack: global memory optimization for stream applications in high-level synthesis. In *Proc. of 22nd ACM/SIGDA Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, pages 233–236, February 2014.

[117] A.V. 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.

[118] Manish Verma and Peter Marwedel. Overlay techniques for scratchpad memories in low power embedded processors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 14(8):802–815, 2006.

[119] R.A. Walker and S. Chaudhuri. Introduction to the scheduling problem. *IEEE Design & Test of Computers*, 12(2):60–69, summer 1995.

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

[121] Joseph G. Wingbermuehle, Ron K. Cytron, and Roger D. Chamberlain. Compiling for power with ScalaPipe. *Journal of Systems Architecture*, 59(8):615–625, September 2013.

[122] Joseph G. Wingbermuehle, Ron K. Cytron, and Roger D. Chamberlain. Optimization of application-specific memories. *Computer Architecture Letters*, April 2013.

[123] Joseph G. Wingbermuehle, Ron K. Cytron, and Roger D. Chamberlain. Superoptimization of memory subsystems. In *Proc. of 15th Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, June 2014.

[124] Joseph G. Wingbermuehle, Ron K. Cytron, and Roger D. Chamberlain. Superoptimized memory subsystems for streaming applications. In *Proc. of 23rd ACM/SIGDA Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, February 2015.

[125] Felix Winterstein, Samuel Bayliss, and George Constantinides. Separation logic-assisted code transformations for efficient high-level synthesis. In *Proc. of 22nd Int'l Symp. on Field-Programmable Custom Computing Machines (FCCM)*, pages 1–8, 2014.

[126] H-SP Wong, Simone Raoux, SangBum Kim, Jiale Liang, John P Reifenberg, Bipin Rajendran, Mehdi Asheghi, and Kenneth E Goodson. Phase change memory. *Proceedings of the IEEE*, 98(12):2201–2227, 2010.

[127] Wm A Wulf and Sally A McKee. Hitting the memory wall: Implications of the obvious. *ACM SIGARCH Computer Architecture News*, 23(1):20–24, March 1995.

[128] Yuan Xie. Modeling, architecture, and applications for emerging memory technologies. *IEEE Design and Test of Computers*, 28(1):44–51, 2011.

[129] Yuan Xie, Gabriel H Loh, Bryan Black, and Kerry Bernstein. Design space exploration for 3D architectures. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 2(2):65–103, 2006.

[130] `http://www.xilinx.com`.

[131] Hsin-Jung Yang, Kermin Fleming, Michael Adler, and Joel Emer. Optimizing under abstraction: Using prefetching to improve FPGA performance. In *Proc. of 23rd Int'l Conf. on Field-Programmable Logic and Applications (FPL)*, pages 1–8, 2013.

[132] Lin Yuan and Gang Qu. Design space exploration for energy-efficient secure sensor network. In *Proc. of Int'l Conf. on Application-Specific Systems, Architectures and Processors*, pages 88–97. IEEE, 2002.

[133] Chuanjun Zhang and Frank Vahid. Using a victim buffer in an application-specific memory hierarchy. In *Proc. of Design, Automation and Test in Europe Conference and Exhibition*, pages 220–225, 2004.

[134] Baihua Zheng, Jianliang Xu, and Dik Lun Lee. Cache invalidation and replacement strategies for location-dependent data in mobile environments. *IEEE Transactions on Computers*, 51(10):1141–1153, 2002.

[135] Ping Zhou, Bo Zhao, Jun Yang, and Youtao Zhang. A durable and energy efficient main memory using phase change memory technology. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 14–23. ACM, 2009.

[136] Omer Zilberberg, Shlomo Weiss, and Sivan Toledo. Phase-change memory: an architectural perspective. *ACM Computing Surveys (CSUR)*, 45(3):29, 2013.

[137] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

Appendix A: ScalaPipe

This appendix is an extension of Section 3.1 and is based on [120] and [121]. As mentioned in Section 3.1, ScalaPipe provides two domain-specific languages (DSLs) in the Scala programming language [85] that are used to generate streaming applications. The *kernel DSL* provides a way to implement processing kernels and the *application DSL* provides a way to connect the kernels together and map them to the target architecture. Because the streaming application is described in DSLs embedded in Scala, Scala language constructs can be used to generate potentially large and complex application topologies and resource mappings.

A.1 Kernel DSL

The ScalaPipe kernel DSL provides a simple imperative programming language that can be used to implement kernels. To use the kernel DSL, one extends the `Kernel` class. Within the kernel DSL, the inputs and outputs for the kernel are specified as well as the implementation. Note that it is possible to use existing kernels in target-specific languages such as Verilog or C. To use such kernels, an `external` statement is used to inform ScalaPipe of the implementation code. It is also possible to mix multiple external implementations for various platforms as well as an internal implementation, which is used if no matching external implementation is available for the desired target.

A.1.1 Language Features

The kernel DSL features many of the standard language constructs available in a traditional programming language for conditionals and looping. By using a version of the Scala compiler with language virtualization features [16], ScalaPipe is able to use the standard Scala control structures such as `if` and `while`. Because of this, much Scala code can be easily reused in ScalaPipe with few changes, which eases the prototyping and testing of kernels.

In addition to the basic control structures and math operators, ScalaPipe provides a rich set of data types, which includes primitive types such as integer, floating point, and fixed point types, fixed length arrays, structures, and unions. Kernels that use only these features can run on any resource that ScalaPipe supports. To allow more flexibility and the ability to interface with library code, ScalaPipe also allows function calls to external libraries and pointers. However, kernels using such features can only be mapped to traditional processors.

A.1.2 Example

```
val AverageU32 = new Kernel {
    val in0 = input(UNSIGNED32)
    val in1 = input(UNSIGNED32)
    val out = output(UNSIGNED32)

    out = (in0 + in1) / 2
}
```

Figure A.1: Example Kernel

A simple example kernel is shown in Figure A.1. This kernel inputs two values from separate input streams and outputs their average on an output stream. First, the kernel will wait for

input to be available from both channels. It then adds the values together and divides by two. Finally, the kernel outputs the average, blocking on the output stream if necessary.

A more complex example for generating pseudo-random numbers using the Mersenne twister algorithm [77] is shown in Figure A.2. This kernel demonstrates several additional features of ScalaPipe, such as local variables and control flow.

In addition to the language facilities that ScalaPipe provides, the Scala language can be used as a type-safe macro pre-processor to enable generic kernel code. Such generic kernels could have compile-type types or other parameters. Further, it is possible to develop kernels that support a compile-time configurable number of ports, as demonstrated by the `GenericSplit` kernel in Figure 3.2.

The code within a kernel can be thought of as executing in a continuous loop. Each time an input port is referenced, a new value is expected and the kernel will wait until input is available if necessary. Likewise, each time an output port is assigned, a new value is produced for the consumer, again blocking if the output queue is full. Kernels that require no inputs run continuously until they execute a `stop` statement or the application terminates.

A.1.3 Intermediate Representation

Before kernel code can be generated, it is necessary to turn the kernel DSL program into an intermediate representation. Since the kernel DSL is embedded in Scala, the issue of parsing is handled by the Scala compiler. The DSL code turns into function calls where the variables in the DSL are actually objects to represent the variables, which are created in the `input`, `output`, and `local` functions. Because variables in the kernel DSL are objects, Scala code and kernel DSL code can be mixed, allowing Scala to act as a macro language. This is similar to *lightweight modular staging* introduced in [97] where variable types determine

```
val MT19937 = new Kernel {
    val out = output(UNSIGNED32)
    val mt = local(Vector(UNSIGNED32, 624))
    val index = local(UNSIGNED32, 0)
    val configured = local(BOOL, false)
    val i = local(UNSIGNED32, 5)  // Random number seed
    val j = local(UNSIGNED32)
    val y = local(UNSIGNED32)
    if (configured) {
        if (index == 624) {
            for(i <- 0 until 624) {
                j = i + 1
                if (j == 624) {
                    j = 0
                }
                y = (mt(i) >> 31) + (mt(j) & 0x7FFFFFFF)
                j = i + 397
                if (j > 623) {
                    j -= 624
                }
                mt(i) = mt(j) ^ (y >> 1)
                if (y & 1) {
                    mt(i) ^= 0x9908b0df
                }
            }
            index = 0
        }
        y = mt(index)
        y ^= (y >> 11)
        y ^= ((y << 7) & 0x9d2c5680)
        y ^= ((y << 15) & 0xefc60000)
        y ^= (y >> 18)
        index += 1
        out = y
    } else {
        mt(index) = i
        i = 0x6c078965 * (i ^ (i >> 30))
        i += index
        index += 1
        configured = index == 624;
    }
}
```

Figure A.2: Mersenne Twister Kernel

whether an expression is executed when the application generator runs or if the expression is compiled into code to be executed later.

Once the abstract syntax tree is built from the kernel code, it is either used directly for code generation or converted to a control flow graph. When generating code for traditional processors or GPUs, the abstract syntax tree is used directly for code generation. However, for FPGAs the abstract syntax tree is first converted to a control flow graph to enable better Verilog code generation.

A.1.4 Code Generation

For traditional processors, C code is emitted. For graphics processors, OpenCL C [107] code is generated. Finally, for FPGAs, Verilog is generated. Since the kernel language maps easily into the C programming language, the abstract syntax tree is used directly for generating code targeted for traditional processors. Unfortunately, generating code for GPUs and FPGAs requires more work.

For GPUs, it is desirable to allow multiple threads to run the same kernel on different data elements. To allow this, ScalaPipe checks the kernel to see if there is state that needs to be preserved across invocations. If there is no state to be preserved, then each element in the input queues can be processed in parallel. In this case, ScalaPipe will generate code to process each item in the input buffer in a separate thread. Note that this simple method for extracting parallelism leaves much to be desired. However, it provides a prototype for evaluating alternative resource mappings. If a higher performance implementation is sought, it is possible to substitute a custom implementation.

Like GPUs, FPGAs present a problem for automatic code generation from an imperative-style language. To generate Verilog, the abstract syntax tree is first converted into a three-

address intermediate representation; that is, most operations can be represented by an operator, a destination, and two sources. These operations are then collected into blocks of operations that can execute simultaneously. This organization allows ScalaPipe to generate state machines where each variable represents a register or wire. Before any of the optimization passes, each operation occupies its own state and all variables are mapped into registers.

As an example of how ScalaPipe generates register-transfer level (RTL), consider the kernel in Figure A.3 for computing the n^{th} term of the Fibonacci sequence.

```
val n       = input(UNSIGNED32)
val result  = output(UNSIGNED32)

val i       = local(UNSIGNED32)
val last    = local(UNSIGNED32)
val current = local(UNSIGNED32)
val temp    = local(UNSIGNED32)

i = n
last = 1
current = 0
while (i > 0) {
    temp = current
    current += last
    last = temp
    i -= 1
}
result = current
```

Figure A.3: ScalaPipe Fibonacci Kernel

Figure A.4 shows the kernel after conversion to ScalaPipe's intermediate representation. Each label represents a state and the `goto` statements indicate state transitions. In some cases, a state may take multiple cycles to complete. This can happen, for example, when waiting for an input, as is the case for state `S1` in Figure A.3, or performing a division

instruction. In such cases, a guard is inserted to prevent the state machine from advancing to the next state until the operating completes.

```
S1:  i = n
S2:  last = 1
S3:  current = 0
S4:  t1 = i > 0
S5:  if t1 then S6 else S11
S6:  temp = current
S7:  current = current + last
S8:  last = temp
S9:  i = i - 1
S10: goto S4
S11: result = current
S12: goto S1
```

Figure A.4: Intermediate Representation of the Fibonacci Kernel

As can be seen from this simple example, this straightforward translation leaves quite a bit of room for improvement. The optimization passes in ScalaPipe attempt to address this issue. The kernel after optimization is shown in Figure A.5. After optimization, the number of states in the state machine has been reduced from 12 states to 4. Note that state `S3` reads and writes to the same variables. This is acceptable because when converted to RTL, all sources will be evaluated before the assignment. Despite additional room for improvement, the current version of ScalaPipe is unable to improve this code any further.

A.1.5 Optimizations

ScalaPipe performs several optimization passes when generating kernels to target hardware. These passes are mostly traditional compiler optimizations, however, some are specific to the goal of generating hardware.

```
S1: i = n
    last = 1
    current = 0
S2: if (i > 0) then S3 else S4
S3: temp = current
    current = current + last
    last = temp
    i = i - 1
    goto S2
S4: result = current
    goto S1
```

Figure A.5: Optimized Fibonacci Kernel

Variable Renaming Variable renaming is an optimization performed by ScalaPipe to expose additional opportunities for other optimizations. In the variable renaming pass, ScalaPipe converts each basic block to static single assignment (SSA) form [31]. ScalaPipe currently does not convert the entire kernel to SSA form to avoid dealing with ϕ functions.

Common Subexpression Elimination Common subexpression elimination (CSE) is a traditional compiler optimization that replaces expressions that are recomputed with the earlier result. Such expressions can be expressed directly in the source program or generated by the compiler when converting the source program into its intermediate representation (for example, math required for array references).

Dead Store Elimination Dead store elimination (DSE) is a compiler optimization that eliminates stores to variables that are not referenced. Such stores are rare since they serve no purpose, but they can show up especially in generic code.

Dead Code Elimination Dead code elimination (DCE) is an optimization that eliminates code that is not used. As is the case with dead store elimination, dead code is rare, but can appear with generic code.

Strength Reduction Strength reduction is an optimization that replaces expensive operations with less expensive operations. For example, division by a power of two can be converted into a shift operation. Although it is possible to write code that already takes strength reduction into account, applying strength reduction as an optimization pass can allow code to be written in a more natural way. Moreover, the values that allow for strength reduction (for example, divisors that are powers of 2) may be present only in some uses of a particular piece of code.

Copy Propagation Copy propagation—another traditional compiler optimization—replaces uses of variables that are the target of direct assignments with their value. For example, given the following code segment:

```
x = y
z = x + 1
```

Copy propagation would yield:

```
z = y + 1
```

Sequences such as these are common in the code generated by the front end of the compiler. Therefore, this optimization is often applicable even if the source program does not contain any such sequences.

State Space Compression State space compression is an optimization that ScalaPipe uses to collapse multiple operations into a single state. ScalaPipe does this by moving all

operations into the earliest possible state. Note that this is equivalent to ASAP scheduling [119]. For example, given the following sequence:

```
S1: a = b + c
S2: a = a - 1
S3: x = y + z
```

State space compression would combine states `S1` and `S3` giving:

```
S1: a = b + c
    x = y + z
S2: a = a - 1
```

Note that states `S1` and `S2` cannot be combined because of a read-after-write data dependency on `a`.

State Elimination State elimination is an optimization that ScalaPipe uses to combine a string of operations into a single state. For example, given the following sequence:

```
S1: a = b + c
S2: a = a - 1
S3: x = y + z
```

State elimination would eliminate the second state by combining it with the first state:

```
S1: a = b + c - 1
S3: x = y + z
```

A.2 Application DSL

The ScalaPipe application DSL is used to connect kernels together and map them to resources. To use the application DSL, one extends the `Application` class.

A.2.1 Overview

The application DSL allows one to specify how kernels are connected using functional composition. Each kernel takes a list of streams for input and returns a list of streams, which can then be passed to other kernels. To allow large and complex topologies to be generated, Scala can be used as a type-safe macro language.

In addition to the application topology, the application DSL is where resource mapping is described. The `map` function is used for this purpose. Given a stream or edge specification as a parameter, the `map` statement marks where data flows from one resource to another.

A.2.2 Resource Mapping

The application DSL creates a graph of kernels connected by streams. Some streams may change resources as indicated by a `map` statement. To map the kernels onto resources, ScalaPipe first assumes all kernels are unassigned and then processes the streams one-by-one until either all kernels are assigned a resource or no more changes occur. If any resources remain unmapped at this point, they are assumed to reside on a traditional processor. Note that it is possible for an invalid mapping to be specified. In this case, the error is reported when the ScalaPipe application generator is run.

Once the resource mapping is complete, ScalaPipe generates the kernels for the required resources and then creates an application with the queues and threads necessary to run the kernels. This application also includes any code necessary for routing data to other devices such as GPUs or FPGAs. For GPUs, ScalaPipe generates code to use OpenCL [107] for communication and compiling of the kernel code. For FPGA devices, code to communicate with the driver for the FPGA device is generated on the software side. On the hardware side, a top level file is generated to connect the kernels on the FPGA device.

A.2.3 Example

```
val app = new Application {
    val rand1 = Random()
    val rand2 = Random()
    Print(AverageU32(rand1, rand2))

    map(AverageU32 -> Print, FPGA2CPU())
}
```

Figure A.6: Example Application

A simple example application is shown in Figure A.6. This application generates two streams of random numbers by instantiating two `Random` kernels. The outputs of these kernels is then averaged using the `AverageU32` kernel described in Section A.1.2. Finally, the output of the `AverageU32` kernel is printed to the screen via the `Print` kernel.

The example also demonstrates a `map` statement. The `map` statement describes the edge between the `AverageU32` and `Print` kernels. The type of edge is an `FPGA2CPU` edge, meaning that this edge connects a CPU resource to an FPGA resource. The result is that all the kernels except the `Print` kernel will be implemented on an FPGA device; the `Print` kernel will be implemented for a general-purpose processor. Each kernel implemented on a general-

purpose processor is assigned its own thread. Note that it is possible to specify multiple FPGA or CPU resources using arguments to the `FPGA2CPU` function.

A.2.4 TimeTrial

To support performance profiling of a streaming application, ScalaPipe has TimeTrial [66] built-in. TimeTrial allows one to instrument the queues between kernels. For example, to discover a bottleneck one might be interested to know which queues are consistently full. The use of TimeTrial in ScalaPipe works much like resource mapping, but with `measure` statements instead of `map` statements.

An example TimeTrial statement for the application in Figure A.6 is shown below.

```
measure(ANY_KERNEL -> AverageU32, 'backpressure)
```

This statement causes ScalaPipe to instrument all edges entering the `AverageU32` kernel. In this case, there are two such edges. For each of these edges, TimeTrial will monitor "backpressure", which is the fraction of time the producer could not enqueue an item to the queue due to the queue being full. This information is reported as the application runs each *frame*, which is typically a 1-second interval of time. Using statements such as this, it is possible to track down bottlenecks in the application.