Where's the Beef? *Why* FPGAs Are So Fast

Scott Sirowy, Alessandro Forin
Microsoft Research

September 2008

Technical Report
MSR-TR-2008-130

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

Where's the Beef? *Why* FPGAs Are So Fast

Scott Sirowy
Microsoft Research
t-scots@microsoft.com

Alessandro Forin
Microsoft Research
sandrof@microsoft.com

Abstract

*Where do all the cycles go when microprocessor applications are implemented spatially as circuits on an FPGA? It is well established that certain sequential applications can be captured spatially and achieve breathtaking speedups when run on an FPGA, but **why**? Despite running at clock speeds orders of magnitude slower compared to their embedded processor equivalents, FPGA applications can "lose" enough cycles to create exceptionally fast spatially-oriented circuits. We profile and analyze three canonical applications amenable to FPGA speedup to quantify exactly where FPGAs gain that speedup. We compare the FPGA implementations to several idealized software platforms. The idealized software platforms give insight as to how FPGA implementations attain such dramatic speedups. We quantify the effects of parallelizing and pipelining instructions, streaming data, and eliminating the instruction fetch, showing exactly where the cycles are lost in an FPGA implementation. We also show how the memory interface to the FPGA will affect the performance. Our results show that custom memory interfaces are the most effective way at enabling much greater performance on the FPGA, and that memory interfaces traditional software use become a bottleneck when the FPGA uses the same interface. The results, though not surprising, provide a clearer and more intuitive understanding of the performance FPGAs can achieve, offering researchers and engineers alike a new angle to attack the task of parallelizing applications.*

1. Introduction

"…Therefore, we were able to attain 10,000X speedup over the fastest software implementation using our novel FPGA implementation …" And so concludes a typical researcher who ported a software application to a field-programmable gate-array (FPGA). Invented in the 1980s, FPGAs are custom computing elements that allow designers to create highly efficient, custom circuit accelerators. In fact, certain application domains have been known to achieve extraordinary speedups when implemented on an FPGA, the results being extensively reported [3][18][21], and with several conferences dedicating forums to FPGA application speedup[9][10].

Qualitatively, the reasons for FPGA speedups are clear. FPGAs can expose parallelism at many different levels, from the bit and instruction level all the way to the loop and task level. A typical von-Neumann computer must fetch an instruction from memory, execute the instruction, and store the result. An FPGA implementation instead thrives on executing multiple (often 10s of) instructions in one clock cycle, and ridding itself of fetching instructions since they are built into the FPGA data path itself. FPGAs implementations can also implement deep pipelines, enabling highly efficient, high throughput circuits that output results every clock cycle. Finally, FPGAs use specialized custom interfaces to memory banks that make the best use of the memory bus for that application.

Quantitatively, the reasons for FPGA have not been completely unfolded. *Why* is it that despite a clock speed often orders of magnitude slower than the microprocessor an FPGA is able to achieve orders of magnitude speedup on the application in both latency and throughput compared to the software implementation? How much speedup can we gain by unrolling a software loop once? Twice? What about pipelining the circuit? What effect does the memory interface have on that pipeline? On the loop unrolling? Quantifying the effects of such optimizations can provide a deeper understanding for applying the same optimizations to other FPGA applications.

In [12], the authors perform an extensive quantitative analysis of FPGA speedups on several simple image processing applications, comparing to baseline MIPS, Pentium, and VLIW platforms. The authors generalize the speedup factors accounted for into a cohesive speedup model, while Dehon [7] develops the concept of computational density to quantitatively analyze the differences between CPUs and FPGAs (as well as ASICs).

We quantify three applications known to achieve high performance when implemented on an FPGA, including an N-body simulation, a JPEG compressor, and an AES decryption algorithm. We do not plan on reporting on their speedup on an FPGA for the sake of speedup, but rather to quantitatively and intuitively show exactly where the speedup comes from when applying well-known FPGA optimizations. We also provide an in depth quantitative analysis on how the memory interface affects FPGA speedups. We quantify and analyze the effects of the FPGA interface by comparing three memory interfaces using three different memory technologies that exist on many FPGA development boards. Our goal is to provide a baseline and quantitative intuition as to where the speedups that FPGAs are known to achieve come from, and a list of practical recipes for attacking the problem of parallelizing an application. The techniques and tools should be valuable with the advent of a more parallel-computing aware generation of researchers and engineers.

The rest of this paper is organized as follows. Section 2 discusses the study methodology. Sections 3, 4, and 5 quantitatively analyze three separate applications and their FPGA implementations. Section 6 quantifies the effects of the speedup attributed to the memory interface. Section 7 concludes.

Figure 1: Analyzed Implementations. The key will serve as a reference for subsequent discussion

	Key	Implementation
SW	BASE	Base MIPS Platform
	S1	BASE + Superscalar Commit*2
	S2	BASE + Perfect Instruction Cache
	S3	BASE + No Instruction Fetch
	S4	BASE+ No I-fetch+ Perfect Data Cache
	S5	BASE+ Superscalar*4 & No I-fetch
	S6	BASE + Perfect Pipeline
FPGA	C1	FPGA data path, No Unrolling
	C2	C1, Loop Unrolled Once
	C3	C2, Loop Unrolled Twice
	C4	C2, Loop Unrolled Four Times
	C5	C1, Pipelined Mem. + Computation
	C6	C2, Pipelined Mem. + Computation
	C7	C3, Pipelined Mem. + Computation
	C8	C4, Pipelined Mem. + Computation
	C9	C1, Fully Pipelined
	C10	C2, Fully Pipelined
	C11	C3, Fully Pipelined
	C12	C4, Fully Pipelined

2. Study Methodology

The goal of this study is to identify exactly why FPGA implementations are so much faster than their software counterparts. For this reason, we chose three applications that are particularly amenable to FPGA speedup. The examples include an N-body simulation, a JPEG compressor, and an AES decryption unit. While each example comes from a different application domain, each one can be characterized as a data-driven, computationally intensive application, enabling efficient FPGA implementations that are suitable and convenient for such an analysis.

We first run the software on a base MIPS architecture based on the eMIPS extensible processor [17]. Each example is run on a highly configurable simulator [11] that features real time hardware and software integrations, and allows the integration of a detailed SRAM and DDR-2 memory model for streaming data to the FPGA. The base MIPS platform (***BASE***) accesses memory through an SRAM interface, requiring five cycles to fetch both instructions and data. The base architecture is not pipelined, and has no instruction or data cache. We used this particular MIPS platform because the platform was simple to analyze, and is characteristic of a platform often used for real-time embedded system applications. To make our study more representative of the general-purpose processor class, we clocked the ***BASE*** at a nominal clock speed of 2 GHz.

To better understand and more intuitively depict why FPGAs attain a much higher performance over their software counterparts, we investigate a spectrum of *idealized* optimizations which augment the base MIPS processor for better performance, shown in Figure 1. While not entirely realistic, the optimizations do give some insight as to why FPGAs are able to attain orders of magnitude speedups on certain applications. Each optimization represents an aspect of execution that FPGAs accomplish very well. The superscalar optimization (***S1***, ***S5***) allows the processor to execute *N* instructions in a single cycle, modeling the ability of an FPGA to exploit instruction level parallelism. We model a perfect instruction cache (***S2***) as an intermediate step in showing the impact of eliminating the execution time required for fetching instructions. Similarly, we model a platform that has no instruction fetch (***S3, S4, and S5***) to more closely model the FPGAs mode of execution, where there is no concept of an instruction fetch. We model a perfect data cache (***S4***) to simulate that an FPGA will probably be reading a large contiguous memory section and streaming data as fast as the memory interface allows. Finally, we model a perfect pipeline (***S6***), allowing the ***BASE*** to complete one instruction per cycle, regardless of actual instruction dependencies or hazards. The perfect pipeline closely models the ability of FPGA applications to complete one data element per cycle, a hallmark of efficient FPGA design for dataflow applications.

Figure 2: N-body Algorithm. Both the loops and the instructions within the loops are independent, allowing an FPGA to exploit a large amount of parallelism

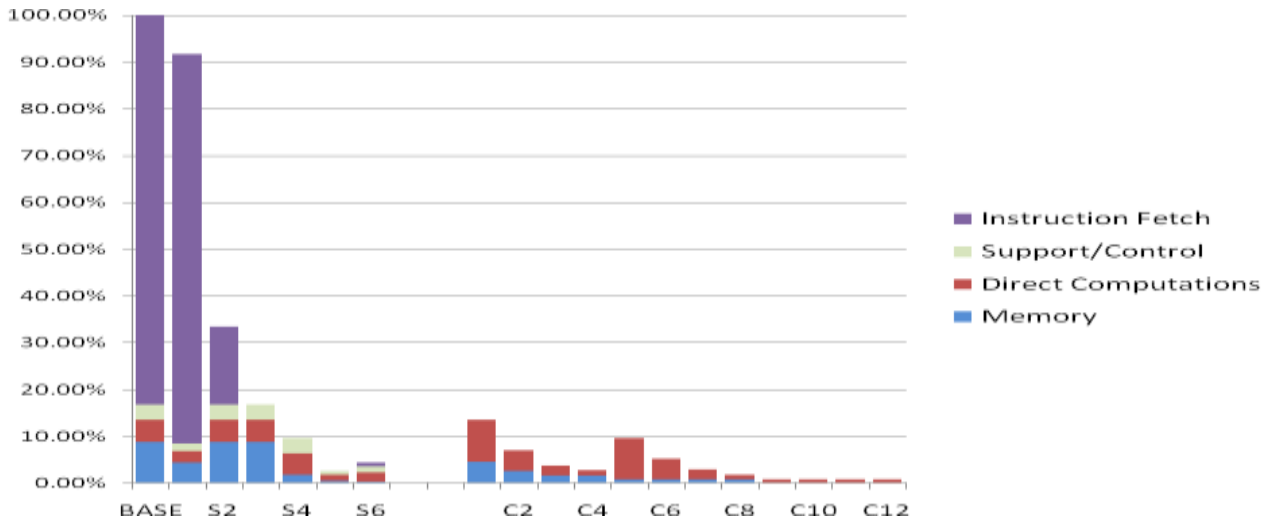
```
For element(i) in space:
    for element(j) in space:
        fx += GravityForce(i,j);
        fy += GravityForce(i,j);
        fz += GravityForce(i,j);
    element(i).fx = fx;
    element(i).fy = fy;
    element(i).fz = fz;
    fx = fy = fz = 0;
```

We compare the software implementations of each example to the FPGA implementations, also shown in Figure 1. For each FPGA implementation, we apply varying levels of well-known FPGA optimizations [6], including loop unrolling (***C1-C12***), pipelining (***C5-C12***), and streaming constructs (***C9-C12***). Where applicable, we manually implemented each of the data paths for the examples in question and synthesized them using Xilinx ISE 9.2 targeting the Virtex-4 LX25 FPGA on the ML401 board[22]. We simulated the circuits using Giano and ModelSim. For the JPEG compression circuit, we utilized extensive data published by [1].

Since the goal of this work was not to report speedups for any given circuit, but rather explain how and why those speedups were attained, we observe that the execution time of each implementation can be broken down into four main elements:

1. Instruction Fetch
2. Support/Control Instructions
3. Direct Computation Instructions
4. Memory Instructions

Figure 3: N-body running on a number of different software platforms and with varying levels of FPGA optimizations.



The first element is the instruction fetch, which accounts for the time required to fetch instructions from memory. The support/control instructions, first noted by [12], are the instructions that do not directly contribute the real computation, but rather update loop counters, branching code, updating the stack pointer, etc. The direct computations account for all the instructions directly responsible for computing the desired results. Finally, the last category is the time taken to execute memory instructions, including loads and stores. Separating and correlating these four elements among the different implementations leads to a more specific breakdown of how the FPGA is able to attain often dramatic speedups.

3. N-body Simulation

One class of applications that are particularly amenable to FPGA speedup is N-body applications. As a broad definition, N-body applications compute the forces and movements for a (often very large) set of interacting particles/bodies using classical mechanics models. N-body applications usually work on a large set of data, and are computationally intensive. The standard particle-particle algorithm is shown in Figure 2. For each element/body in the data set, the N-body algorithm computes the force contributed by each of the other elements/bodies in the data set. The algorithm executes one time step, and then updates each body's position according to the bodies' velocity and the forces previously computed. The naïve particle-particle algorithm runs in $O(n^2)$, although more advanced algorithms can compute a time step in $O(nlog(n))$, using approximation for bodies beyond a certain threshold. While using optimized software might seem like a good starting point for creating customized FPGA applications, recent research has shown that is not always the case [20] and that often the most naïve algorithm produces the best circuits for FPGA execution.

Our particular implementation of N-body operated on 500 distinct bodies, but the analysis would have been similar for both larger and smaller data sets. We first ran the resulting N-body application on a number of software platforms in Figure 1. The distributions are shown in Figure 3. Each column reports the execution time as a percentage of the base platform (***BASE***).

Figure 4: N-body. Computations on each force element

	Implementation	Cycles per Force Element
SW	Base MIPS (BASE)	900
	Perfect I-Cache (S2)	408
	Superscalar*4 & No I-fetch (S5)	60
	Perfect Pipeline (S6)	123
FPGA	FPGA data path (C1)	94
	Loop Unroll 1 (C2)	49
	Full Pipeline (C9)	4

We broke down the execution time for each application run into four separate elements: instruction fetch cycles, direct computation cycles, memory cycles, and support instruction cycles. The base platform (***BASE***) spends 87% of its time fetching instructions, and another 9% accessing data memory. The last 4% of the ***BASE***'s execution time is split between direct computation instructions and support instructions. As shown in Figure 4, the ***BASE*** requires 900 cycles to compute the force for one element in the space. A 2-way superscalar platform (***S1***) is not much better compared to the ***BASE***. Since instruction fetches dominate the execution time, only 13% of the entire time is amenable to speedup via a superscalar platform. ***S1*** did achieve a 7% speedup compared to the base platform. When we supplement the ***BASE*** with a perfect instruction cache (***S2***), we begin to see a more balanced distribution and larger speedups. ***S2*** only spends 57% of its time fetching instructions, since fetching only takes one cycle. Data memory accesses now account for 29% of the execution time, direct computations take 8%, and support/control instructions account for 6%. The fourth platform models an idealization of a processor that does not have an instruction fetch (***S3***). While not realistic, the modeling technique does give some insight as to where speedup from an FPGA implementation comes from since an FPGA has no

concept of an instruction fetch. ***S3*** runs 7.5X faster than the ***BASE***. Memory accesses dominate, accounting for 69% of the execution time. The computations that relate directly to the force computation take 18% of the time, and the support/control instructions account for the final 13% of the execution time. We can improve the execution time by supplementing the previous ideal platform with a perfect data cache (***S4***), allowing our platform to effectively read data from memory in one cycle. The resulting platform attains ~17X speedup. Direct computations now account for 41% of the execution time, support/control instructions take 28% of the time, and memory accesses take 31%. The resulting execution profile is beginning to resemble an FPGA implementation. A perfect four-way superscalar platform (***S5***) achieves 62X speedup over the ***BASE*** because it is able to execute four computations in parallel. This results in 39% of the execution time coming from direct computation execution, 27% of the execution from support/control instructions, and memory accesses occupying 35% of the execution time. Finally, we run the N-body application on a model of a perfect pipeline (***S6***). ***S6*** runs ~44X faster than the base platform. Memory computations take 8% of the execution time; direct computations take 42% of the time; support/control instructions take 30%, and the instruction fetch takes 20% of the execution time.

By executing the N-body application on a number of different idealized platforms, we gain insight into how the FPGA gains performance for the equivalent software platform. The superscalar models showed they can attain performance gains by parallelizing at the instruction level. Platforms without an instruction fetch immediately showed FPGAs gain performance because they have no concept of an instruction fetch. Similarly, FPGA implementations gain speedup by eliminating or hiding the support/control instructions, including branch instructions, updating loops, and moving data because of a lack of registers. Finally, FPGAs can gain by pipelining operations.

We now analyze several FPGA implementations for the N-body inner loop, which calculates the force between two bodies in the simulation. We implemented an N-body circuit that had a latency of 61 cycles to compute one force element. The circuit could be clocked at 251 MHz. The right hand side of Figure 3 shows the distributions for FPGA circuit implementations with various levels of loop unrolling and pipelining. Figure 4 shows some of the FPGA implementations and how many total cycles they require to compute one force calculation. All of the circuit implementations were able to eliminate the cost of fetching instructions *and* the cost of the control instructions. The N-body circuit allows controlling software to push data as fast as it can through the N-body circuit, eliminating the need for any control flow within the circuit. The leftmost bar (***C1***) in Figure 3 shows a single FPGA data path implementing the N-body force calculation. The circuit fetches data from DDR memory, and accounts for 34% of the circuit's execution time. Therefore, 66% of the time is spent calculating the force. Figure 4 shows that an FPGA data path is able to complete one force calculation in 94 cycles. The computational latency of the data path is only 61 cycles, but accessing the DDR-2 memory requires another 33 cycles. Still, the FPGA data path is ~7X more efficient than the base MIPS processor. Part of the FPGAs efficiency is due to the elimination of the instruction fetch and control cycles (which account for 89% of the base platform's time), but the FPGA is also able to schedule multiple instructions in parallel, which accounts for ~2X efficiency after the instruction fetch and control instructions are eliminated. Factoring in a slower clock speed of 251 MHz (~8X slower than the MIPS platform), the FPGA data path is able to achieve ~7X compared to the ***BASE***.

The FPGA implementation can be further improved by unrolling the inner loop a number of times, exhibited by the next few bars (***C2-C4***) in Figure 3. Loop unrolling once, twice, and four times results in speedups of 13X, 25X, and 35X respectively, compared to the ***BASE***. As shown in Figure 4, loop unrolling once reduces the number of clock cycles to compute one force element to 49 cycles. The latency of the data path is still the same, but the FPGA can compute two forces at the same time. The distributions in Figure 2 show us that the FPGA increases the amount of computations performed in one time step by loop unrolling. The percentage of time spent on computation reduces from 66% with the base FPGA data path to 63% when we unroll the force loop once, 58% when the loop is unrolled twice and 40% when the loop is unrolled four times. Unrolling the calculation shifts the bottleneck from the calculation to the data fetch. In ***C4***, the memory accesses account for 60% of the execution time.

We can improve upon the naïve versions of loop unrolling by pipelining the accesses to memory while the previous calculation is taking place such that the data is ready for the next execution. Pipelining the memory accesses (***C5-C8***) for each of the four loop unrolling examples results in a ~1.2-1.4X speedup compared to the original loop unrolling implementations (***C1-C4***). The majority of the execution time is now concentrated on the computation, 92% for ***C5*** and 60% for ***C8.***

We fully pipeline the N-body calculation into a 61 stage pipeline (***C9***), allowing the circuit to achieve speedups of ~127X compared to the ***BASE***. A full pipeline can complete one force calculation every four cycles, 225X more efficient than the ***BASE***. Extra time must be spent paging and refreshing the DDR-2 memory. Those extra cycles actually cause the pipeline to stall several times during execution. Unrolling the pipelined circuit (***C10, C11, and C12***) does not offer any additional opportunities for speedup since the 32-bit bus is completely occupied supplying just one pipeline with data. The N-body circuit could have certainly also performed better with a more custom memory interface to allow for larger bandwidth, but the current analysis was fixed to a development board that only had a 32-bit interface to memory. In Section 6, we quantitatively analyze how the memory interface to the FPGA affects the amount of speedup the FPGA achieves for the N-body simulation.

4. JPEG Compression

Image processing and compression algorithms are also well suited for FPGA implementations. The Joint Photographic

Figure 5: JPEG Compression Steps.

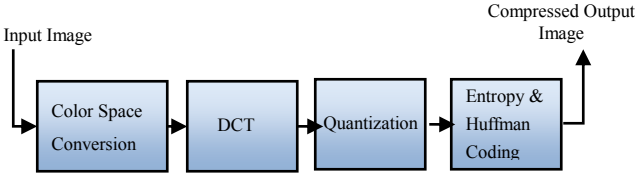
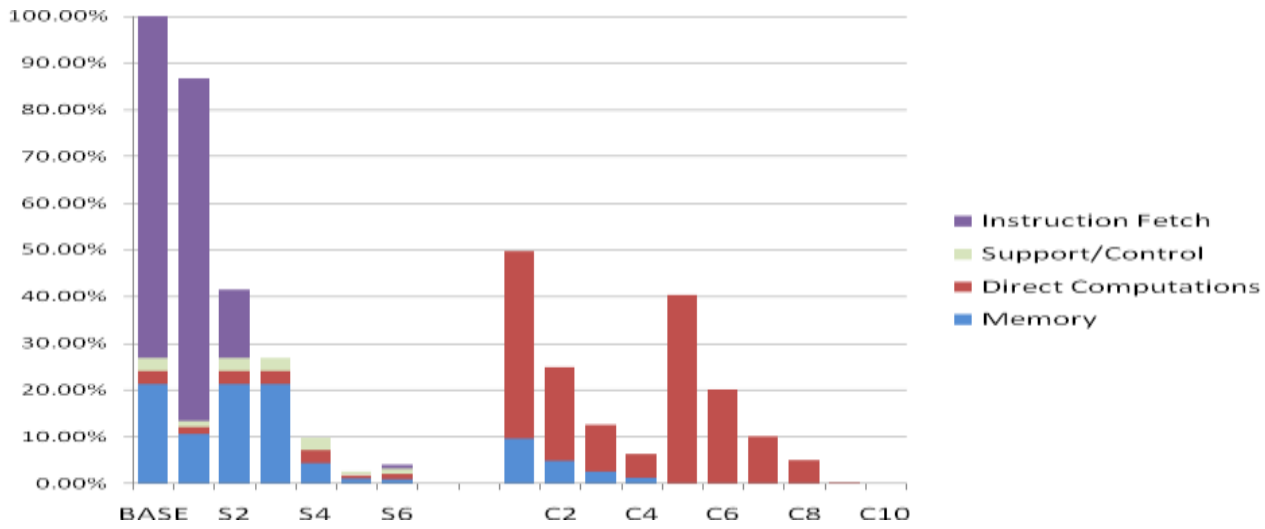


Figure 6: JPEG Compression running on a variety of different software platforms and FPGA implementations.



Experts Group proposed the JPEG compression standard in 1987[16], a lossy compression algorithm that can compress an image by orders of magnitude. A number of different compression techniques exist [4] based on the standard, which involves many different operation modes to control lossiness, etc., but most techniques involve several distinct steps, highlighted in Figure 4.

A high level diagram of the JPEG compression algorithm is shown in Figure 5. A JPEG compressor begins by first transforming an image from the red, green, blue (RGB) color space to the *YCbCr* color space, where *Y* is the brightness of a pixel, and the *Cb* and *Cr* elements represent the chrominance of a pixel. The second step involves performing the discrete cosine transform on each 8x8 block of pixels in the picture. The picture is then quantized, eliminating many of the high frequency elements. The encoding concludes by running entropy coding and Huffman encoding on the quantified data.

Each step in the JPEG compression standard is independent from the rest, offering an opportunity to take advantage of task level parallelism not seen in the N-body simulations. Further, each step within the compression process (besides the final encoding stage) operates independently on either each pixel, or a block of pixels, allowing an FPGA to further exploit both instruction level and loop level parallelism.

Figure 6 shows the distributions for running the JPEG compression algorithm on a 1024x768 sized image, both in software and as FPGA circuit implementations. The analysis is similar for different sized images. We first examine the left side of Figure 6, which shows the JPEG compression algorithm running on the base MIPS platform as well as the idealized optimized MIPS architectures. The ***BASE*** spends 75% of its time fetching instructions from memory, 22% fetching from and storing to data memory, and a pale 3% on both direct computations and support/control instructions. Compared to the N-body example, the JPEG compression algorithm shows a higher dependence on memory operations. The compression algorithm fetches 1MB worth of pixels, and uses main memory for intermediate storage. Still, again we see that a significant portion of the time is spent fetching instructions from the processor's memory. Figure 7 shows that the ***BASE*** spends 1155 cycles per color component of a pixel. Therefore, the ***BASE*** requires 3465 cycles to process one pixel. As we progress down the list of software and circuit optimizations, we'll see how this number can be improved greatly. As with the N-body application, running the compression algorithm on ***S1*** only improves the execution by a minute 14% since the real bottleneck is the instruction fetch and data memory accesses. ***S2*** makes a steady 2.5X improvement over the base processor by allowing instruction fetched to only take one cycle. As shown in Figure 7, computing one color component only requires 568 cycles. While the 2.5X improvement might be misleading since there is only a ~2X difference between the two cycle counts, we recall that the memory is operating on a much slower clock, so when the instruction cache eliminates 2X of the cycles, the overall speedup is even more. ***S3*** reaches just over 4X speedup. ***S3*** is thus left with a memory bottleneck where 89% of the execution time is fetching from and storing to data memory. ***S4*** more closely models streaming data to an FPGA. ***S4*** balances its time more efficiently, leading to a ~14X speedup. ***S5*** is able to compute a color component in 62 cycles, leading to a 55X speedup over the ***BASE. S5*** is still spending 60% of its time accessing memory, but the other 40% is split evenly between direct computations and the support/control instructions. The perfect pipeline's (***S6***) execution time is distributed such that 20% of the time is spent on fetching instructions, 23% of the

Figure 7: JPEG. Computation per color component

	Implementation	Cycles per Color Component
SW	Base MIPS (BASE)	1155
	Perfect I-Cache (S2)	568
	Superscalar*4 & No I-fetch (S5)	62
	Perfect Pipeline (S6)	146
FPGA	FPGA data path (C1)	300
	Loop Unroll 1 (C2)	149
	Full Pipeline (C9)	1

time spent on data memory accesses, and 28% for both the direct computation and support/control instructions, resulting in ~50X speedup. Again, we see that the perfectly pipelined machine is actually slower than the platform with a lot of instruction level parallelism and no instruction fetch (***S5).***

The right half of Figure 6 shows several FPGA implementations of the JPEG compression algorithm. The FPGA data path is based on the work of Agostini et. al. and synthesizes to a maximum clock frequency of 160 MHz, or 12X slower than the ***BASE***. The authors created a component for each task in the compression process, allowing parallelism at several different levels. When we do not consider pipelining as the authors did, the FPGA data path (***C1***) is able to achieve nearly a 2X speedup compared to the ***BASE***. The data path latency is 243 cycles, leading to a distribution where the computations take 80% of the time, and the memory operations take 20%. As noted in Figure 7, even though the data path finishes the computation in 242 cycles, the total number of cycles required is 300. The extra cycles are due to the overhead of fetching and storing to DDR-2. Since ***C1*** does not stream, a lot of cycles are wasted fetching from memory. Comparing to the ***BASE***, ***C1*** eliminates the instruction fetch, eliminating 75% of the execution time. Despite the slower clock speed, ***C1*** further improves its execution time by eliminating or hiding support/control instructions, which account for a nominal 3% of the ***BASE***'s execution time. The data path also requires 70% fewer memory cycles per pixel (including instruction fetch) than the ***BASE. C1*** does not improve on the direct computations cycles of the ***BASE*** since the data path thrives at working on multiple pixels at a time. Still, the data path improves on the ***BASE*** by ~2X.

Loop unrolling (***C2-C4***) becomes advantageous for compressing multiple images, or working on multiple 8x8 blocks within the same image. Loop unrolling once (***C2***) results in just over a 2X speedup over ***C1***, and 3.9X compared to the ***BASE***. The streaming of multiple data elements accounts for the non-linear speedup. The loop unrolled data path suffers from having a clock speed over 12X slower than the ***BASE*** clock, but the ability to execute multiple blocks in parallel overcomes the deficiency. In this particular case, the FPGA gains speedup by eliminating both the instruction fetch and the support/control instructions, which together account for 78% of the ***BASE*** implementation. The FPGA streams data from memory in bursts, fetching data 85% more efficiently than the ***BASE***. ***C2*** also takes advantage of both instruction and task level parallelism to operate on each pixel in 5X fewer cycles than the ***BASE***. Figure 7 shows that ***C2*** only needs 149 cycles to compute a color component. While the latency of the data path is still the same, the data path can compute two at a time resulting in ~2X speedup. The cycles have decreased slightly because of the ability to fetch two words of data at the same time.

We can pipeline (***C9-C10***) the JPEG compressor, allowing one color component to be input every cycle and a JPEG word to be output every cycle. Being able to input a pixel every three cycles results in a 568X speedup over the ***BASE***. Compared to the perfect pipeline software implementation (***S6***), the circuit is 11X faster. Let us examine the comparison between the two pipelined implementations. Once again, the clock speed of the circuit is over 12X slower than the microprocessor implementation, putting the FPGA circuit at an immediate disadvantage. Even though software is able to effectively commit one instruction/cycle, the software implementation is only able to complete the computation on a color component in a pixel every 146 cycles, which is .6% as efficient as the hardware pipeline. Eliminating the 28% of the time the perfect pipeline spends on control instructions means the perfect pipeline still requires 100 cycles per pixel component. The numbers suggest that a tremendous amount of speedup is coming from being able to execute a number of computations in parallel. Combining the component speedups, the overall speedup is 11X faster than a pipelined MIPS implementations, and 568X faster than the ***BASE***.

Since the pipelined FPGA data path (***C9***) only requires eight bits of information per cycle, the 32-bit bus allows the FPGA to compress four images concurrently, resulting in ~2100X speedup compared to the ***BASE***. The fully pipelined circuit can compute one color component per cycle, or 1155X faster if we analyze the cycle counts, compared to the ***BASE.*** If each JPEG image is completely independent from the other, the speedups from each pipeline are additive (almost, DDR-2 takes several penalties when accessing a lot of data), whereas software implementations must compress each image in turn.

5. Advanced Encryption Standard (AES)

The Advanced Encryption Standard (AES), also known as Rijndael, is described in [5]. AES is a block cipher, allowing the algorithm to operate on a fixed length set of bits. In contrast to the N-body and JPEG compression examples, FPGA AES implementations exploit parallelism at the bit level, instructions level, and the loop level. Figure 8 depicts a high level diagram of the encryption algorithm. The inputs are a 128-bit plaintext block and a one-time input of a 128-bit key. The first step is to replace every byte in the plaintext block by its substitute in what is known as the *S-box*. In the second step, the rows are shifted in a defined manner. The third step involves mixing the columns by performing a number of XOR operations on the rectangle representation of the 128-bit block. Finally, the result is *XORed* with the sub key, which is an intermediate generated result based on the original key. The process is actually repeated nine times for a 128-bit key. AES's applicability to FPGA speedup has spawned a number of groups developing efficient circuit implementations, trading off both size and performance [8].

We analyze a software AES decryption algorithm running on 128-bit blocks. The AES decryption algorithm is equivalent to the encryption algorithm in complexity, and the same analysis applies. The input to the AES decryption algorithm was a page

Figure 8: AES Algorithm. Encrypting/Decrypting a 128-bit block actually involves performing the following process for nine rounds.

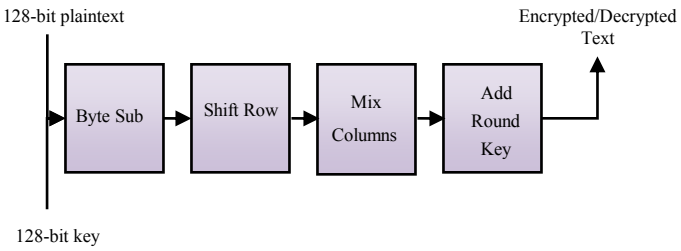
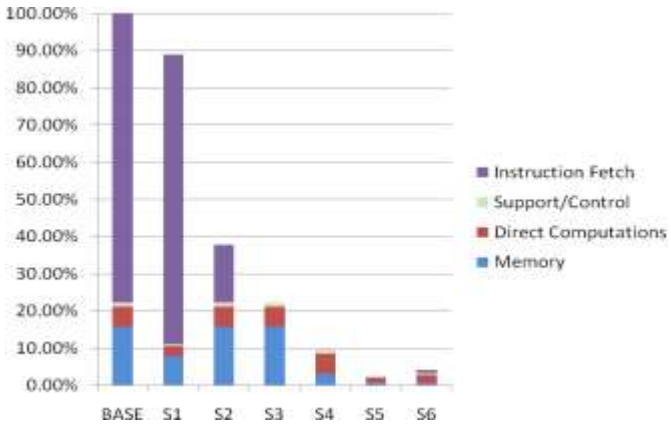
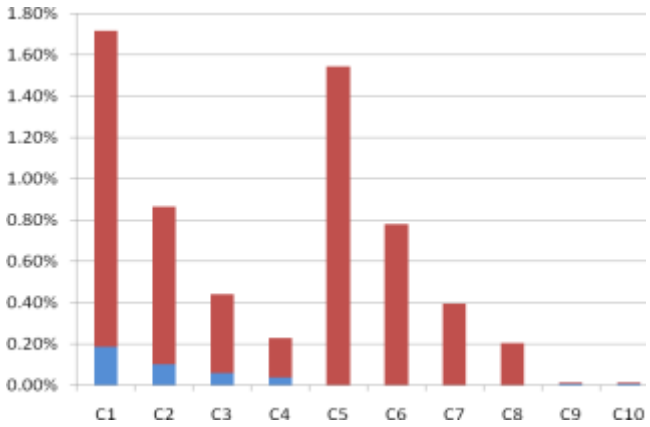


Figure 9: (a) AES running on different software platforms. Eliminating the instruction fetch speeds up AES by ~5X (b) Different FPGA implementations





worth of encrypted text (204 blocks of 16 bytes each). The distributions for both the software and FPGA implementations are shown in Figure 9. Figure 10 shows how many cycles several implementations require to encrypt one AES block. The ***BASE*** spent 80% of the execution time fetching instructions, 16% accessing data memory, 3% on direct computation instructions, and only 1% on support/control instructions. The results were expected since the decryption algorithm has to fetch a large amount of data before executing the algorithm. The decryption algorithm also uses memory to store the S-box lookup table values. Overall, the ***BASE*** required 89,438 cycles to encrypt one AES block. ***S1*** only improves the ***BASE*** execution time by 11% because the instruction fetch still dominates the execution time. ***S2*** required just less than half the cycles (41,438) to encrypt one AES block, leading to a 3.6X speedup over the ***BASE. S3*** speeds the decryption time up by ~5X compared to the ***BASE***, showing that the lack of an instruction fetch poses one of the best opportunities for speedup on an FPGA. ***S4*** performs ~15X better than the ***BASE***. This platform attempts to model the ability to stream data to the FPGA as fast as possible. Still, improvements can be made. ***S5*** runs AES ~60X faster. At this stage, the majority of the time is still fetching memory at 49%, but the computation time has jumped to 42% of the time. The support/control instructions still occupy 9% of the time, which shows room for improvement with an FPGA implementation. The ***S5*** model resembles the FPGA's ability to schedule multiple instructions in parallel. Figure 10 shows that ***S5*** encrypts an AES block in 5,285 cycles. ***S6*** achieves a 46X speedup over the ***BASE***. The direct computation time jumps to 53%, the support/control instructions take 11% of the time, memory instructions take 16% of the time, and the instruction fetch takes 20% of the time. Assuming we could eliminate the instruction fetch and support instructions from the perfect pipeline, and execute some of the direct computation instructions in parallel, the final implementation would closely resemble a very efficient FPGA implementation.

Figure 9 (b) shows the speedups and distributions for the FPGA implementations. The FPGA data path operates at the maximum clock frequency of 116MHz, a factor of 17X slower than the ***BASE***. ***C1*** operates on only one 16-byte block yet performs remarkably well, attaining 56X speedup over the

Figure 10: AES. Computation per 128-bit block

	Implementation		Cycles per AES Block
SW	Base MIPS (BASE)		89438
	Perfect I-Cache (S2)		41438
	Superscalar*4 & No I-fetch (S5)		5285
	Perfect Pipeline (S6)		11760
FPGA	FPGA data path	(C1)	566
	Loop Unroll 1	(C2)	286
	Full Pipeline	(C9)	2

BASE. We can also see in Figure 10 that ***C1*** only requires 566 total cycles to encrypt one block. The data path generates decrypted text in 504 cycles, while the memory interface adds 63 more cycles... Since there is no instruction fetching and/or control instructions, the FPGA data path spends 11% of its time accessing memory, and 89% of the time computing on the decrypted text. Compared to the ***BASE, C1*** speeds up the memory operations by 1140X, and the computational instructions by 37X. Using Amdahl's law and factoring in the FPGA AES clock speed (116MHz), the overall speedup is 56X.

Loop unrolling offers more opportunity for speedup. Loop unrolling can mean decrypting different blocks within the same encrypted message, or different encrypted messages altogether. Duplicating the loop once (***C2***), twice (***C3***), and four times (***C4***) results in 111X, 217X, and 409X speedup, respectively, over the ***BASE***. An analysis of ***C3*** shows that 14% of the time is spent accessing memory, and 86% is spent on computation. Loop unrolling twice requires four times the amount of data, and thus the memory bandwidth requirement increases, but not linearly since the FPGA can stream data on both the rising and falling edges of the clock. ***C3*** has longer memory latency since four data paths require data, but the memory streams the data, resulting in more efficient memory accesses. Despite fetching data from the same 32-bit DDR interface, the circuit can now stream data 400X more efficiently than software platforms can access memory since they have to multiplex between memory, support, and direct computation instructions.

FPGA implementations can further hide the memory latency behind the previous computation stage. The next four bars (***C5-C8***) show that by pipelining the memory stage with the computation such that data is always ready for the computation, the FPGA can achieve 123X, 242X, and 458X speedups over the main software implementation when the main loop is unrolled once (***C6***), twice(***C7***), and four times(***C8***). The pipelining of the memory accesses gains an average 11% over loop unrolling alone (***C1-C4***). Further, AES can be fully pipelined. A fully pipelined AES circuit can reach over 4000X speedup over the ***BASE***, effectively completing a 128-bit block each clock cycle. In contrast, the ***BASE*** requires almost 19,000 cycles to complete the same task.

6. Quantification of the FPGA Memory Interface

One of the most important and often overlooked elements of FPGA speedup (or slowdown) is the memory interface to the FPGA. The previous sections have shown that FPGA implementations thrive by being able to exploit parallelism at many different levels, including the instruction and loop levels. But, if the FPGA application requires more data in a given cycle than the memory interface can allow, the FPGA will suffer a latency penalty for accessing too much data from the memory. For example, the N-body data path requires eight 32-bit elements per cycle to perform one computation. The memory interface only allowed for two data elements a cycle, thereby reducing the throughput of the data path from one computation per cycle to one computation per four cycles (for the pipelined implementations). The analysis also showed that duplicating the data path multiple times resulted in no extra speedup over just one pipelined N-body circuit. This was precisely because the memory interface was not even large enough to effectively handle one pipeline, let alone multiple.

In this section, we quantify the effects of the memory interface on the FPGA speedup. We will concentrate our analysis on the N-body example presented in Section 3. The resulting analysis is similar for the other examples. We compare running the N-body simulation from three forms of memory on the FPGA platforms, DDR-2 RAM, SRAM, and onboard Block RAMs (BRAM). The first type of memory is commonplace in PCs and is now appearing on many new FPGA boards due to its economy of scale and price. The second type is favored in embedded systems due to its predictable response times. It is more expensive and uses more area than DDR memory chips. The third type is unique to FPGAs and many embedded chips that provide "scratchpad" memories of one type or another. It is available in the most limited quantities and the price is hard to assess since it is built into the FPGA or embedded processor. The ML401 board provides 64 MB of DDR memory, 1 MB of SRAM memory, and up to 162 KB of BRAM. There are many other types of memories with different properties, but our analysis of these three basic types should suffice to illustrate the most relevant performance issues.

Figure 11: DDR-2 RAM Specification Summary.

Speed	266 MHz
Bus Width	32 bits
Bank Size	64 MB
# of Banks	1
RAS	28 cycles
Burst Length	4 or 8
Page Boundary	8 kB
Active to Refresh	70 ns
Refresh All	7.8 us

The DDR-2 RAM specifications are summarized in Figure 11. The DDR-2 RAM has a 32-bit interface and runs at 266 MHz. DDR-2 (Dual Data Rate) reads two data elements per cycle, one on the rising edge of the clock, and one on the falling edge, thus doubling the DDR's given bandwidth. Because it can burst data in packs of four or eight 32-bit data elements, the DDR-2 RAM can stream a large amount of data assuming the data is contiguous in memory. We assume the data is stored in

Figure 12: Execution Times(s)(*on a log scale*) for N-body using three different memory interfaces, one with DDR-2, one with SRAM, and one with BRAM.

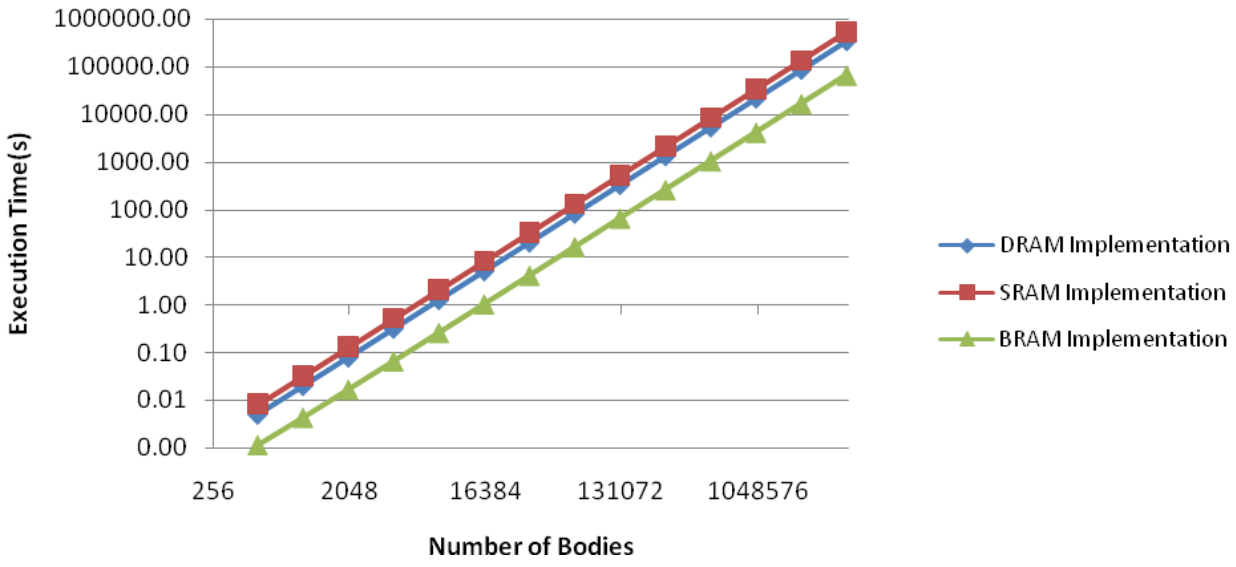
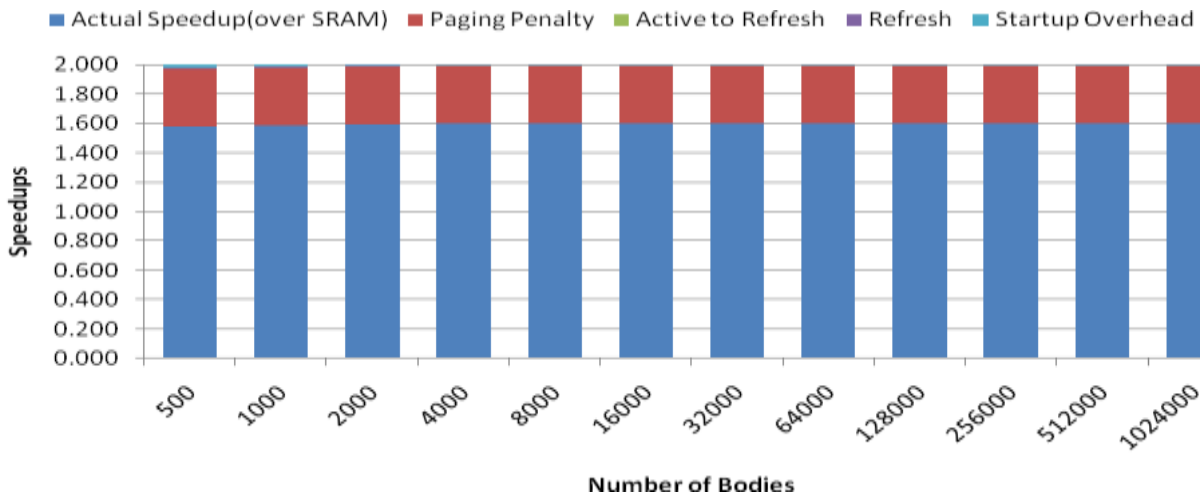


Figure 13: Analysis of SRAM and DDR-2 Interfaces. DDR-2 should attain 2X speedup, but several factors prevent it from approaching the ideal speedup.



memory contiguously to allow the memory controller to take advantage of the streaming data capabilities. DDR-2 technology is built using one transistor and one capacitor for each bit of storage, meaning at specific time intervals, the memory must be refreshed. A more thorough explanation of how dynamic RAM works can be found at [19]. Consequently, DDR-2 must be refreshed every time the memory controller accesses a new page, and on two specific timing intervals, including the active to pre-charge time interval and the refresh all time interval. The penalty for refreshing is 28 cycles, symbolized by the row access strobe (RAS) in the figure for the DDR-2 [13] on our development platform. We use only one bank of 32-bit memory, but a designer that plans to target an FPGA might take advantage of an even more custom memory interface with more banks to both reduce the refresh penalties and offer more memory bandwidth.

The second memory on the FPGA platform is the SRAM. The SRAM also runs at 266 MHz, and has a memory latency of 5 cycles. But, as with the DDR-2, the SRAM can also stream data using a burst command and a smart memory controller. The memory interface to the SRAM is 32-bits, allowing one 32-bit data element per cycle, or half the bandwidth of the DDR-2. However, the SRAM does not have to ever refresh itself due to the technology on which it is built upon [19].

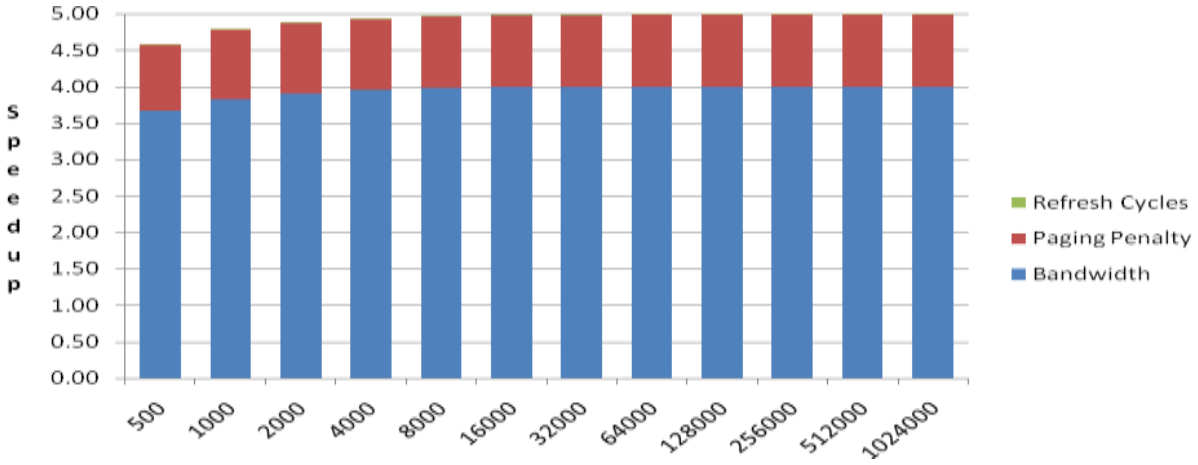
The third memory technology we focus our quantitative analysis on is block RAM (BRAM). BRAMs are small distributed RAMs located throughout the FPGA fabric that allow the designer to create custom, high bandwidth solutions at compile time. Thus, a designer can easily create a large 128-bit, 256-bit, etc. size bus for the particular application at hand. One drawback of BRAM is there is often little storage (162 KB of data storage vs. 64 MB for DDR-2). With a smart memory controller, a designer could also potentially stream data from the DDR-2 to the BRAM interface to give the application much greater bandwidth.

For the purposes of quantitatively assessing speedup, we only concentrate on performance of the three memory technologies in creating custom memory interfaces, but there are certainly tradeoffs to utilizing any of the three in a particular implementation, including area, power, and size.

Figure 12 shows the execution times for running the same N-body application from Section 3 with DDR-2, SRAM, and BRAM interfaces. We varied the number of bodies run in the simulation from 512 to 1,024,000, running on a fully pipelined version of the FPGA data path (***C9***). As expected, the BRAM implementation was able to deliver the most memory to the FPGA data path per cycle, resulting in a 7.2X-8X speedup over the SRAM implementation, and 4.5X-5X speedup over the DDR-2 memory interface. Similarly, the N-body implementation with the DDR-2 interface runs approximately 1.57X-1.6X faster than the N-body FPGA application running with an SRAM interface. As we did for the different application implementations, we break up the elements of speedup for each of the memory interfaces.

We first look at the elements of speedup comparing the SRAM and DDR-2 interfaces to the N-body circuit, shown in Figure 13. Ideally, given that both DDR-2 and SRAM have the same bus width, and both run at the same clock frequency, DDR-2 should be able to attain 2X speedup over the SRAM interface because of DDR-2's ability to fetch two memory words every cycle. Due to the refresh and startup overheads, the DDR-2 is not able to achieve the ideal speedup figures. We broke up the actual speedup into several elements, including *paging penalty*, *active to refresh* transitions, *refresh all*, and *startup overhead*. The *paging penalty* is the cost of accessing memory that spans the page boundary of the DDR-2. As shown in Figure 11, the DDR-2 has a page boundary of 8 KB and if all of the application data can be kept within one page, there is no *page penalty* for accessing data on a different page. Every time the application pages, the application must take a paging penalty equal to the time to access a new row (RAS), or 28 cycles. The *active to refresh* time refers to the fact that an active row must be refreshed at a given interval to ensure the memory data stays

Figure 14: Analysis of BRAM and DDR-2 Interfaces. A BRAM interface is able to achieve 5X speedup over a DDR-2 interface even though the bandwidth is only 4X greater. The rest of the speedup comes from paging penalty and refresh penalty cycles.



valid. In this case, the active to refresh time is 70ns. The *active to refresh* also means that we must activate a row again at the RAS time interval. Finally, all the rows in a DDR-2 must periodically be refreshed, meaning that we must activate the current row again once the refresh has taken place. The *refresh all* time models this penalty in the overall execution time.

Figure 13 shows the DDR-2 speedup over SRAM as a fraction of the ideal speedup of DDR-2 over SRAM. For most of the execution runs, the *paging penalty* accounts for 39% of the slowdown compared to the ideal speedup accounted for on bandwidth alone. For the smallest example, startup overhead accounts for 1% of the slowdown. The *startup overhead* disappears for all but the smallest examples since the memory controller does not need to activate the DDR-2 memory. The *active refresh* and *refresh all* overheads account for less than .5% of the slowdown. Combining the 2X bandwidth speedup and the slowdowns associated with *paging penalty* costs and *refreshing* the memory at dedicated intervals, the DDR-2 still performs consistently better than SRAM wherever the examples stream a large amount of contiguous memory.

We provide a similar analysis showing how a BRAM interface attains such a large speedup compared to a DDR-2 interface for the N-body simulation. As stated earlier, the BRAM interface is able to achieve a 4.5X-5X speedup over the DDR-2 implementation mostly by using a much larger bus width, but the speedup alone does not really explain the whole story. Figure 14 shows how a BRAM interface is able to achieve much greater speedups than the DDR-2 interface. For each example, we created a custom 256-bit interface to the FPGA data path by combining the BRAMs into one wide bus. We distributed the data into the BRAM such that a separate BRAM memory was connected to a data path input. Thus, we could completely fill the FPGA pipeline and deliver a new result every cycle once the pipeline is filled. Figure 14 shows that roughly 80% of the speedup comes from the fact that the BRAM has can deliver more data to the FPGA data path than DDR-2 can. Recall that the bus width of DDR-2 is 32 bits but can deliver two words per cycle. The BRAM thus has a 4X greater bus width, accounting for most of the speedup. The smaller examples on the left show that the data path doesn't actually get 4X speedup on bandwidth alone. This is because the data path still hasn't been completely utilized. In most of the examples though, the bandwidth accounts for 4X of the speedup. As we saw in the analysis of DDR-2 and SRAM interfaces, BRAM attains 19% more speedup because the DDR-2 frequently has to page, adding cycles that neither the BRAM nor the SRAM suffer. For large examples, the *paging penalty* pushes the BRAM speedup over DDR-2 to 5X. The refresh penalties, including the *active to refresh* time and the *refresh all* times, add an insignificant amount of speedup. Because the N-body application is streaming data from a contiguous memory, the *paging penalty* completely dominates any notion of refreshing at dedicated intervals.

7. Conclusion

We analyzed how FPGAs attain so much speedup over their sequential software counterparts. Our goal is to provide researchers and engineers a quantitative intuition how to attack the problem of parallelizing certain applications, both for software and FPGA platforms. We presented an extensive quantitative analysis of three different applications, a baseline N-body simulation, a JPEG compressor, and an AES decryption algorithm. By first showing how each application ran on both a baseline MIPS processor and several ideal optimized software architectures, *and* breaking up the execution time into several elements, we created a spectrum that visually and intuitively showed how an FPGA gathers speedup. The superscalar models showed how FPGAs can exploit instruction level parallelism. Platforms without an instruction fetch closely modeled the fact that FPGAs have no instruction fetch phase. We then compared those ideal optimized platform executions to several FPGA implementations, with varying levels of loop unrolling and pipelining. We showed how FPGA implementations lose cycles by eliminating the software instruction fetch, hiding the control instructions, executing multiple instructions in parallel, and pipelining those instructions. We then quantitatively showed

how the memory interface to the FPGA affects the performance. We showed that custom interfaces enabling high bandwidth enable much greater speedups, and that current development boards will always have a memory bottleneck for large FPGA applications. There are penalties from using DDR memory, which is the most appealing from a price-area-capacity viewpoint. The penalties can be limited to about 20%, but only when the data is arranged in very large and contiguous blocks. BRAMS are the easiest and most flexible way to create large and custom memory busses, which are very effective means to achieve speedups up to 8X over SRAM and ~5X over DDR. The very limited capacity of BRAMS means that they must be used in conjunction with some other external memory for most practical applications.

References

[1] AGOSTINI, L., B. SERGIO., AND SILVA, I. High Throughput Architecture of JPEG Compressor for Color Images Targeting FPGAs. ICECS 2006.

[2] ALTERA CORPORATION. http://www.altera.com

[3] BEECKLER, J. S. AND GROSS, W. J. 2005. FPGA Particle Graphics Hardware. FCCM 2005

[4] BHASKARAN, V. AND KONSTANTINIDES, K. Image and Video Compression Standards Algorithms and Architectures Second Edition. Kluwer Academic Publishers, USA, 1999.

[5] DAEMEN, J. AND RIJMEN, V. The Design of Rijndael. AES-The Advanced Encryption Standard. Springer 2002.

[6] DE MICHELI, G. Synthesis and Optimization of Digital Circuits. McGraw Hill Higher Education. 1994

[7] DEHOHN, A. The Density Advantage of Configurable Computing. IEEE Computer, vol. 33, No.4, April 2000

[8] ELBIRT, A.J., YIP W., CHETWYND, B. AND PAAR C. An FPGA Implementation and Performance Evaluation of the AES Block Cipher Candidate Algorithm. AES Candidate Conference. 2000.

[9] FCCM. Field-Programmable Custom Computing Machines Conference. http://www.fccm.org

[10] FPGA. International Symposium on Field-Programmable Gate Arrays. http://www.ece.wisc.edu/~kati/isfpga/

[11] FORIN, A. , NEEKZAD, B. ,AND LYNCH, N. Giano: The Two-Headed System Simulator. Microsoft Technical Report?

[12] GUO, Z., NAJJAR, W., VAHID, F., AND VISSERS, K. 2004. A quantitative analysis of the speedup factors of FPGAs over processors. FPGA '04

[13] INFINEON TECHNOLOGIES. HYB25D256 256-Mbit SDRAM Data Sheet. January 2003. Version 1.1

[14] LIENHART, G., KUGEL, A. AND MANNER, R. Using Floating Point Arithmetic on FPGAs to Accelerate Scientific N-body Simulations. FCCM 2002.

[15] MEYER, K. AND HALL, G. Introduction to Hamiltonian Dynamical Systems and the N-body Problem. Springer Publishing 2002.

[16] PENNEBAKER, W. AND MITCHELL, J. JPEG Still Image Data Compression Standard. Van Nostrand, 1992.

[17] PITTMAN, R.N, LYNCH, N., AND FORIN, A. eMIPS, a Dynamically Extensible Processor. Microsoft Technical Report. October 2006.

[18] TSOI, K. H., LEE, K. H., AND LEONG, P. H. 2002. A Massively Parallel RC4 Key Search Engine. FCCM

[19] VAHID, F. AND GIVARGIS, T. Embedded Systems Design: A Unified Hardware/Software Introduction. Wiley Publishing 2001

[20] VILLEREAL, J. AND NAJJAR, W. Compiler Hardware Acceleration of Molecular Dynamics Code. FPL 2008.

[21] WHITTON, K., HU, X. S., YI, C. X., AND CHEN, D. Z. 2006. An FPGA Solution for Radiation Dose Calculation.FCCM

[22] XILINX, INC. http://www.xilinx.com