Toward a Hardware Accelerated Future

Toward a Hardware Accelerated Future

A dissertation presented

by

Michael John Lyons

to

The School of Engineering and Applied Sciences

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

in the subject of

Computer Science

Harvard University

Cambridge, Massachusetts

June, 2013

Dissertation advisors
David M. Brooks, Gu-Yeon Wei

Author
Michael John Lyons

Toward a Hardware Accelerated Future

Abstract

Hardware accelerators provide a rare opportunity to achieve orders-of-magnitude performance and power improvements with customized circuit designs.

Many forms of hardware acceleration exist—attributes and trade-offs of each approach is discussed. Single-algorithm accelerators, which maximize efficiency gains through maximum specialization, are one such approach. By combining many of these into a many-accelerator system, high specialization is possible with fewer specialization limits.

The development of one such single-algorithm hardware accelerator for managing compressed Bloom filters in wireless sensor networks is presented. Results from the development of the accelerator highlight scalability deficiencies in the way accelerators are currently integrated into processors, and that the majority of accelerator area is consumed by generic SRAM memory rather than algorithm-specific logic.

These results motivate development of the accelerator store, a system architecture designed for the needs of many-accelerator systems. In particular, the accelerator store improves inter-accelerator communication and includes support for sharing accelerator SRAM memories. Using a security application as an example, the accelerator store architecture is able to reduce total processor area by 30% with less than 1% performance overhead.

Using the accelerator store as a base, the ShrinkFit framework allows accelerators to grow and shrink, to achieve accelerated performance within small FPGA budgets and efficiently expand for more performance when larger FPGA budgets are available. The ability to resize accelerators is particularly useful for hybrid systems combining GP-CPUs and FPGA resources, in which applications may deploy accelerators to a shared FPGA fabric. ShrinkFit performance overheads for small and large FPGA resources are found to be low using a robotic bee brain workload and FPGA prototype.

Finally, future directions are briefly discussed along with details about the production of the robotic bee helicopter brain prototype.

Contents

List of Figures

List of Tables

Previous work

Portions of this dissertation appeared in the following:

S. Chang, A. Kirsch, and M. Lyons. Energy and storage reduction in data intensive wireless sensor network applications. Technical Report TR-15-07, Harvard University, 2007.

M. J. Lyons and D. Brooks. The design of a bloom filter hardware accelerator for ultra low power systems. ISLPED, Aug. 2009.

M. J. Lyons, M. Hempstead, G.-Y. Wei, and D. Brooks. The accelerator store framework for high-performance, low-power accelerator-based systems. IEEE CAL, 9(2): 5356, 2010.

M. J. Lyons, D. Brooks, and G.-Y. Wei. The accelerator store: A shared memory framework for accelerator-based systems. ACM TACO, 8(4):122, 2012.

M. Lyons, G. Wei, and D. Brooks. Shrink-fit: A framework for flexible accelerator sizing. IEEE CAL, PP(99), 2012.

M. Lyons, G. Wei, and D. Brooks. The ShrinkFit Acceleration Framework: Simplifying Multi-Accelerator System Development. Under review.

Acknowledgements

The past seven years have hardly been spent alone. I'm thankful for the many people who have been with me on this journey.

I would like to thank my family, who have shared my ups and downs many times over. To my parents, thank you for your unwavering trust, advice, and love. To my brother and sister, thank you for being there for me, with plenty of cake. And to the rest of my family, thanks for your support and understanding throughout the many Thanksgivings where I "just had to code up one more thing." I couldn't have gotten this far without your love and patience.

I must thank my friends in Boston, who have kept me grounded each day. Nathan and Hallie, your friendship means more than words or ice cream can express. Sasha and Mike, thank you for the tapir-positive company, over waffles and espressos, for the past ten years. Jud and Kevin, thank you for hearing me out, hanging out, and goading me on day after day.

I certainly couldn't do this without the support of the lab. David and Gu, thank you so much for your years of guidance and advice. Mark, Ben, and Ankur, thank you for leading the way and helping me find my feet when I was new to graduate school. Glenn, thanks for your patient help at all hours! And thank you to my labmates, for their countless hours of discussion, opinion, and tolerance. And good luck to those of you who are still working toward their degree and those who are just starting. If you ever need anything, don't hesitate to ask.

I'd also like to thank my friends at Dropbox, who helped me power through to the end. In particular, I'd like to thank Tido, Alex, Sujay, and Arash, who gave me the energy for my last push.

And finally, I'd like to mention two people who I met while in the midst of my PhD. Nora and Ben, you fill me with hope, and I look forward to all that you will accomplish in the future.

This work is dedicated to Ben and Nora

Chapter 1

The potential for accelerated computing

For many years, the computing industry mined clock scaling for consistent performance improvements. Eventually, rising power demands put an end to clock scaling, forcing the industry to switch to multicore architectures. This pivot has not been a silver bullet: multicore performance improvements have failed to keep pace with previous clock scaling gains. The computing industry needs a third approach, not just to regain, but to exceed the performance improvements once taken for granted. A growing body of research suggests that approach is hardware acceleration.

Hardware accelerators (circuits customized for a specific workload or class of workloads) offer many advantages over multicore approaches. First, acceleration can boost performance up to 100x-1000x [30, 33], far beyond multicore's ideal linear increases. Second, acceleration is not dependent on parallelized code and offers significant single-threaded speedups. Third, acceleration performance is not threatened by hard power budgets and the "dark silicon" problem, which will force greater portions of processors to be unused.

By specializing for a specific workload, accelerators can realize orders-of-magnitude performance improvements. This specialization is a double edged sword, however, and results in logic that can only perform its specialized task. The trade-off between performance and workload flexibility, as well as scalability, design complexity, and production cost, distinguish the many types of accelerated computer architectures. No single accelerated architecture is best for all needs—several

approaches exist each with unique trade-offs. This chapter surveys many of these accelerated architecture families, and examines the benefits and trade-offs of each.

This chapter continues with Section 1.1, which discusses the computer industry's previous sources of performance improvements, and describes why acceleration is poised to be a significant part in the future. Section 1.2 analyzes many popular approaches to acceleration in research and in production. Section 1.3 discusses current and upcoming research challenges, critical to realizing the accelerated computing vision.

1.1 Trends in computing

Decades of computing have produced countless innovations. The computing industry once depended on process scaling to simultaneously improve performance through faster clock speeds and keep power consumption manageable. When threshold voltage scaling slowed and increased clock speeds incurred prohibitive amounts of power, the computing industry looked to multicore approaches for an alternative source of performance increases. Although the performance of individual cores do not improve much, the increasing core counts on chip offer additional performance improvements. However, multicore performance improvements have not kept pace with performance improvements once offered by clock scaling. A third approach, hardware acceleration, is necessary.

This section explains how the computing industry got here, why current multicore approaches do not keep pace with necessary performance improvements, and how acceleration can meet or exceed performance improvements once taken for granted.

1.1.1 The end of clock scaling

Moore's Law, which states the size of a transistor shrinks in half every eighteen months, has been the cornerstone of processor improvements for most of the industry's existence. Performance improved when process technologies shrunk, since each transistor became faster and more transistors could fit on the same piece of silicon. Threshold voltages also shrank with successive process technologies, keeping power budgets from exploding despite faster clock speeds and doubling transistor counts.

Unfortunately, performance improvements ran dry near the turn of the century. Although process technologies continued to shrink, threshold voltage scaling slowed. Clock speeds could

continue scaling upward, but the power cost would exceed 100W, beyond which operation and cooling costs become too great [11].

1.1.2 The limits of multicore

Although process technology scaling ceased to provide significant threshold voltage reductions, process scaling continued to offer more transistors for the same piece of silicon. To make the most of the increasing transistor counts, multicore architectures utilizing multiple copies of the same cores gained favor. Ideally, processor performance increases linearly with each additional core. However, several impediments dampen the performance increases once common with clock scaling:

Parallelism. Most software cannot take advantage of multicore architectures automatically, but instead must be parallelized to use multiple cores at once. In many workloads, the additional work required to parallelize software is difficult, or parallelization will not improve performance due to multithreading overheads. Even for most parallelized software, there is a limit to parallelism, after which performance suffers from diminished returns. For many workloads, there is a point past which more cores will not substantially increase performance.

Energy. Despite curbing clock frequency increases, multicore architectures are beginning to hit energy limits again. Due to the slowing of threshold voltage reductions and exponential increase in transistors, some multicore processors will not be able to utilize all of their logic simultaneously. Up to 91% of some processors may need to be turned off by the 11nm node, resulting in large amounts of unused logic called "dark silicon" [56].

Slow clocks. Before multicore approaches, successive processor designs utilized both increased transistor budgets and faster clock speeds. The additional transistors were used for more complicated logic, such as deeper pipelines, more complex branch predictors, and so on. Although multicore architectures make use of increasing transistor counts, multicore's performance gains from increased transistors (1.35x) cannot keep up with voltage scaling era performance improvements from faster clock speeds *and* more transistors (1.58x) [35].

1.1.3 Hardware acceleration

Hardware acceleration (logic optimized for specific workloads) offers an alternative source of performance improvements. Rather than depending on faster transistors or more transistors, hardware acceleration uses transistors more efficiently for certain tasks. As a result, hardware acceleration does not suffer from the multicore problems mentioned above. Accelerators can provide up to and beyond 100x performance improvements [29, 33] without relying on faster clock rates or parallelized software. In addition, performance improvements can be exchanged in part or in full for reduced energy through voltage and clock scaling.

Hardware acceleration typically relies on three types of optimizations to improve performance, energy, and power:

Private memories. Accelerators often contain private memories, which are only used by that accelerator. This allows the accelerator to access memory without contention and to use wider memories for additional memory bandwidth.

Custom operations. Support for custom operations allows accelerators to do complex calculations quickly, rather than relying on a general purpose (GP) processor to execute tens or hundreds of instructions.

Fine-grained parallelism. Accelerator hardware can be explicitly designed to do multiple calculations and access multiple private memories without the rigidity of SIMD or overheads of multithreading.

Dark silicon remains when using accelerators, but is not problematic for many accelerated systems. Each accelerator may only be used for a certain task and turned off at other times. For example, many smartphones currently use hardware acceleration to decode audio files, but only turn the accelerator on when playing music. Under this accelerated approach, only a small portion of accelerators will be on at a time, so large portions of the chip silicon are already dark and will not require shutting down active regions of the processor.

The biggest downside to hardware acceleration is the trade-off between customization and workload flexibility. Greater acceleration is usually possible with more specialization, but this trade-off means that the accelerator will be usable for fewer types of workloads or classes of workloads.

Unlike general purpose CPUs (GP-CPUs) which support any workload, hardware acceleration only targets a smaller subset. Therefore, acceleration is well suited for common workloads that will realize large performance, energy, or power benefits, but not occasional tasks.

Accelerators offer the potential to regain or even exceed past performance improvements, but there is little agreement on how to build accelerated systems. Several accelerated architectures are used commercially, and research efforts produce new approaches regularly. This paper surveys proven and proposed architectures, and identifies several promising directions for accelerated architecture research.

1.2 Accelerated architecture taxonomy

Accelerated architectures come in many forms. This section discusses several approaches for integrating acceleration into computing architectures. This discussion includes an overview of each accelerated approach and discusses six evaluation attributes:

Execution improvement considers performance, power, and energy improvements. The three improvements are closely related because performance speedups can be traded off for power and energy reductions by reducing clock frequency and supply voltage.

Customization describes the design variation within the architecture. This attribute spans from completely homogeneous designs supporting many workloads and consisting of several copies of the same core, to completely heterogeneous architectures containing many unique cores, each designed for a narrow set of workloads. Customization also reflects the portion of the processor expected to be active at any given time. Distributing work across multiple cores is much easier in homogeneous designs since each core can process each workload with equal skill. Heterogeneous systems tend to utilize less of the processor at any given time because only certain cores are capable of performing a particular workload. Due to growing power density and the dark silicon problem, reduced utilization may not be problematic.

Scalability reflects the architecture's ability to increase compute logic. The architecture's interconnect, which connects different components within the system, is particularly crucial for architecture scalability.

Table 1.1: Summary of analyzed accelerated architectures

Execution improvement is expressed in magnitudes. Reconfigurable computing design complexity is both low (FPGA fabric design) and high (logic implemented in FPGA).

	Execution improvement	*Customization*	*Scalability*	*Design complexity*	*Production cost*	*Target audience*
Digital signal processor	10x	Low	Medium	Low	Low	Medium
Graphics processing unit	100x	Low	High	Medium	High	Medium
Specialized homogeneous multicore	100x	Medium	High	Medium	Medium	Medium
Customized ISA	10x	Medium	High	Low	Low	Low
Single-ISA	1x	Low	Medium	Low	Low	High
System-on-Chip	1000x	High	Low	High	Medium	High
Reconfigurable computing	10x	High	High	Low/high	High	Low

Design complexity describes the additional design complexities created by the accelerated architecture. This includes accelerator and system hardware design complexities as well as operating system and application software design complexities.

Production cost considers the additional monetary cost to fabricate a processor designed with the accelerated architecture. This cost is highly correlated with silicon area.

Target audience indicates the audience size for a processor designed with the architecture. An architecture that is more flexible to varying workloads can target a wider audience than an architecture targeting a specific workload.

Table 1.1 summarizes these six attributes for each of the accelerator architectures discussed in the remainder of this section.

1.2.1 DSP

Digital signal processors (DSPs) are microprocessors modified to efficiently analyze signals. These modifications provide:

Parallelism through widened VLIW or SIMD instructions.

Tight looping to keep the pipeline full for repetitive tasks.

Customized ALU support for common DSP operations including fast multiplication.

Multiple memory accesses in the same cycle to enable common multiply operations at every cycle [24].

When writing DSP software, designers first use the C programming language to quickly develop code and remain close to the processor's execution semantics (pointers, heaps, stacks). Python, Java, and other higher level languages might result in faster software development, but would create too much abstraction between the programmer and the DSP hardware. In contrast, assembly language gives the programmer full access to the DSP's hardware, but slows software development to a crawl.

DSP code optimizations are often too complex for C compilers. These optimizations require absolute control over code design, and are performed in assembly language. DSP designers will first profile their program to identify commonly run code, known as "hotspots." These hotspots are run the most often, and through optimization offer the biggest performance improvements.

DSP programmers implement these optimizations by replacing hotspot C code with hand-tuned assembly code. To write optimized assembly code, DSP programmers must have a deep knowledge of the DSP processor's capabilities and timing. This knowledge usually only applies to specific DSP processors, and limits software platform independence.

DSP processors are available from several vendors, including Analog Devices, Texas Instruments (TI), and STMicro. DSP cores are also integrated into some mobile processors, such as TI's OMAP family [7].

Execution improvement

DSP processors are known to improve performance on the order of 10x when compared to GP-CPUs [69]. Although DSPs do not typically implement any unique power saving features, DSP performance improvements can be traded off for reduced power and energy consumption via frequency and voltage scaling.

Customization

Although DSPs feature a fully functional ISA, they are optimized for algorithms making frequent use of multiplication in tight loops. This makes DSPs an excellent choice for specific workloads, such as audio and video media processing. DSPs are highly homogeneous, and share much with GP-

CPU design. As a result, most if not all of the DSP's transistors are utilized when the processor is active. The biggest distinguishing factor of DSPs is multiply optimization in ALUs and parallelized (VLIW, SIMD) instructions.

Scalability

Although a single DSP could scale the number of ALUs upward, the complexity of such an approach would quickly eliminate performance gains. Instead, DSP cores could be replicated, much like general-purpose homogeneous multicore processors. This approach would closely resemble graphics processing unit (GPU) architectures.

Design complexity

DSP architecture complexity is comparable to general purpose microcontrollers and microprocessors. The major differences include wider instructions, a greater emphasis on looping, multiple memory ports, and additional ALUs optimized for multiplication.

Software development is more complicated, since it requires a detailed knowledge of the DSP timing behavior and optimization in assembly language. Due to the need for applications to run "close to hardware," DSP operating systems typically resemble real time operating system (RTOS) kernels [6] and are simple compared to common GP-CPU operating systems.

Production cost

DSPs are roughly the same size as corresponding GP-CPUs. The customizations do not add much to the area cost.

Target market

DSPs are typically limited to signal processing applications. Popular signal processing applications include audio, video, wireless, and speech analysis.

1.2.2 Graphics Processing Unit (GPU)

Graphics processing units typically contain more than 100 simple, identical cores. These cores tend to be good at processing data in easily divided, parallelizable datasets. Each core has limited

memory bandwidth, and applications must take care to to limit the bandwidth requirements of each core. Cores are designed to efficiently perform single precision floating point calculations, which are common in graphics processing. Software is most efficient when identical copies of the same code are run on each core, processing different data sets, via SIMD. GPU core architecture is as simple as possible, and struggles with control flow-heavy workloads.

GPUs use multiple memory levels, similar to memory caching levels on GP-CPUs. At the highest level, GPUs typically provide a shared cache, which is visible to all cores in the GPU. This shared structure can quickly become a bottleneck if overused, so efficient GPU applications tend to rely on local caches (visible to one core) for most memory needs. GPU programming requires explicit control over data transfers to distribute work to these local caches, and to recover the results of computation afterward.

GPUs are most commonly used for rendering graphics for video games, and are predominately manufactured by ATI and Nvidia. Software toolkits such as OpenCL and CUDA have been instrumental for accelerating scientific domain workloads.

Execution improvement

Performance improvements of 50x are typical when porting GP-CPU workloads to GPUs, and acceleration of 400x has been reported [68]. Of course, these improvements apply only to workloads well suited to SIMD parallelization and with low per-core memory bandwidth needs.

GPUs are typically performance focused, and currently are less concerned with energy or power savings. Power management approaches, such as multiple voltage domains or sub-second dynamic voltage and frequency scaling (DVFS) are limited to research, but not yet available.

Customization

GPUs are homogeneous, consisting of hundreds of copies of the same simple core. For this reason, all of the cores are utilized used when the GPU is active.

Scalability

GPUs have already demonstrated the ability to scale into the hundreds of cores. The least scalable aspect of this architecture is the global shared memory interconnect. As the number of cores

continues scaling upward, the per-core bandwidth to the global memory will proportionally decrease, and become a tighter bottleneck. In addition, data must be copied between private GPU core memories and the computer's DRAMs over the PCI bus. PCI and DRAM bandwidth limitations may also limit GPU performance in the future.

Design complexity

Although each GPU contains hundreds of cores, each core is an identical copy, simplifying hardware architecture design. Software design becomes more complicated however, since workloads must be parallelized, memory bandwidth requirements minimized, and data transfers carefully scheduled for hundreds of cores. Some of these complexities are lessened by toolkits, such as OpenCL and CUDA.

Production cost

GPU silicon area can be large, and some designs consume 3 billion transistors [26]. Roughly half of GPU area is used for cores, and the remaining half is used for graphics-specific logic.

Target market

GPUs are limited to easily parallelized workloads with low memory transfer needs. The largest markets with these workload characteristics include video game graphics and domain science.

1.2.3 Specialized homogeneous multicore (SHM)

Specialized homogeneous multicore architectures have much in common with computers utilizing a GP-CPU and GPU. In both approaches, one general purpose CPU controls a large number of identical, simple cores. The SHM architecture replaces graphics rendering acceleration with acceleration for other workloads:

Anton uses acceleration for molecular dynamics to accelerate drug discovery simulations by 100x [67].

Rigel use a hierarchically connected set of 1000 simple, GPU-like cores to accelerate multiple algorithms with a 100x order of magnitude performance improvement[42].

SODA uses four DSP-like cores with software defined radio (SDR) customizations to improve $\frac{performance}{power}$ by roughly 100x [47]. Additional SDR customizations in the Scotch architecture update resulted in additional improvements [76].

Cell uses eight "SPE" floating-point, SIMD cores to accelerate media, gaming, and scientific workloads by roughly 200x [60].

Execution improvement

Typical $\frac{performance}{power}$ improvements using the SHM architecture are in the range of 100-200x, as the examples above illustrate. Many architectures, specifically Rigel and SODA, trade off performance improvements to lower power consumption.

Customization

Like GPUs, SHMs depend on limited memory transfer bandwidth and SIMD-friendly workloads. Some SHM processors are limited to certain domains (Anton, SODA), and others widen workload support by supporting more generic workloads (Rigel, Cell). Market demand for these processors range from the very limited molecular dynamics audience targeted by Anton, to the wide consumer market for the Cell processor. Although SHM processors contain two types of cores (the GP-CPU controller and the highly replicated compute cores), almost all work is done by the replicated cores, which are identical. In this sense, SHM processors are highly homogeneous, so work can be freely distributed between compute cores and sustain high silicon utilization across the SHM processor when active.

Scalability

SHM design scalability varies anywhere from four to over 1000 cores. Designs with few cores use simple, flat interconnects. The shared bus and ring topologies utilized by SODA and Cell, respectively, do not scale effectively in the hundred core regime. However, hierarchical interconnects of clusters, like those used by Anton and Rigel, offer scalability into hundreds and thousands of cores.

Design complexity

SHM architecture design complexity is on the same order of a GPU design. The bulk of the work remains in the design of the compute cores, which can be quite complicated when using larger cores such as Cell and SODA. SHMs utilizing greater numbers of simpler cores also require the additional system design complexity of a scalable interconnect. The controller GP-CPU is unlikely to add much hardware complexity, since most designs use off the shelf implementations including ARM or PowerPC designs.

Software design complexity is also similar to GPUs. The programmer must support two types of processors and must also partition workloads and processing techniques to execute well on multiple, memory bandwidth limited cores. There may be additional complexities implementing compiler tools for the computation cores, as they are custom designed.

Production cost

SHM production cost can be quite varied, depending on the size and number of computation cores. The cell processor contains roughly 300M transistors, though many-core designs such as Anton and Rigel may reflect similar area attributes due to the large number of cores on chip.

Target market

The size of the target market for SHM processors varies highly from design to design. On one end of the spectrum, Anton is highly customized and useful only to the small molecular dynamics audience. On the other end of the spectrum, the Cell processor is widely used for the Playstation 3 and scientific compute clusters.

1.2.4 Customized ISA (CISA)

The customized ISA approach integrates specialization into the GP-CPU, rather than creating distinct accelerator cores. This approach results in new instructions that are limited to finer grained acceleration at the instruction level. Of course, the quantity of additional instructions is limited by the instruction bit width, especially for 32-bit ISAs. Due to the fine-grained nature of the custom ISA approach, adding software support for customized instructions is mostly automatic, courtesy

of the compiler.

Tensilica's Xtensa processor kit is widely used for the development of customized 32-bit RISC processors and corresponding software toolchains [27]. Xtensa-based processors have been used in well-known projects, including Hameed, et al. [30] and Anton. The Conservation Cores architecture takes customization one step further by automatically analyzing source code, identifying kernels ripe for acceleration, and synthesizing customizations for the core [72].

Execution improvement

Tensilica reports performance improvements of 4-72x. As with other architectures, performance enhancements can be substituted for reduced energy and power consumption.

Customization

The customized ISA architecture approach is moderately flexible. Unlike the coarse-grain of accelerator blocks found in SoCs, CISA's acceleration is smaller, on the instruction level. As a result, any algorithm that can benefit from the single instruction's acceleration can benefit from CISA performance improvement. However, custom instructions may still only apply to a small percentage of common computing tasks and are hardly "general purpose."

Because acceleration is built into a GP-CPU, much of the customized core consists of general purpose logic. From this perspective, the percentage of the processor spent on customization is quite low, and often only consists of a few thousand additional gates. Also note that using a CISA core to implement SHM cores, as Anton does, will result in a mostly homogeneous collection of the same replicated CISA core.

Scalability

Scalability of customized ISA architectures can be considered both in terms of custom instructions per core, and multicore scalability. Scaling custom instructions within the core is currently limited by the instruction set size, although this problem will lessen with a transition to 64-bit ISAs. Although a CISA core could contain hundreds of customizations, only one customized instruction can be executed at a time. For this reason, there is little reason to scale the number of customizations in a single core.

An alternative technique is to use multiple CISA core designs. Current approaches have synthesized a homogeneous collection of the same CISA core on a single processor. This approach has proven to scale into the thousands of cores using hierarchical interconnects. A heterogeneous collection of CISA cores could also be used to increase the number of parallel accelerated instructions. However, the duplication of non-accelerated logic inherent when replicating GP-CPU cores may be a downside to homogeneous or heterogeneous multicore CISA scaling. The general purpose portion of the CISA core constitutes the majority of logic, and replicating CISA cores to increase customization comes at a significant overhead.

Design complexity

Developing a CISA core is relatively simple, because only customizations for an already existing GP-CPU core need implementing. Software development also tends to be low complexity, because the GP-CPU's compiler only needs to be modified for the new customizations. Application support for customizations can be automatic via recompilation, assuming the modified compiler recognizes opportunities to use the new instructions. Because CISA is integrated into the GP-CPU core, no explicit memory transfers are necessary unless building a multicore CISA processor.

Production cost

Core customizations typically require only thousands of extra gates, and do not require a significant increase in area or production costs.

Target market

CISA cores are typically customized for a specific workload, so each CISA core's audience is significantly limited.

1.2.5 Single-ISA heterogeneous multicore (SIHM)

Single-ISA heterogeneous multicore processors combine multiple cores that implement the same or similar ISAs. This approach allows for multiple implementations of a core that favor different strengths and weaknesses. Kumar, et al. uses cores from successive Alpha families to build a SIHM containing simple and lightweight cores to the latest, most complex cores [44]. By switching to

advanced cores only when the workload provides a benefit, the SIHM approach reduces unnecessary active logic.

Execution improvement

The SIHM approach does not attempt to increase performance, but to maximize power efficiency. The set of cores contained in a SIHM processor are usually obtained off the shelf, rather than designed specifically for the SIHM processor. Since one core is used at a time, there is no opportunity to improve performance using existing designs. Rather, SIHM allows simpler cores, without branch prediction for example, to be used when workloads would not see performance gains for more complex cores. This approach can reduce power and energy consumption by roughly 40%.

Customization

SIHM's reliance on a single-ISA limits the customization possible. Any customizations unique to a single core must not modify the ISA, which limits the amount of customization. Although an ISA customization could be included in all cores, this would result in a large design effort and since the customization would be common on all cores, would no longer offer any trade-off by switching cores.

Relying on off-the-shelf GP-CPUs further halts customization. By their general-purpose nature, all of the cores are designed not to optimize any particular workload.

Scalability

SIHM processors could scale to many cores, though it is unlikely that many cores supporting the same ISA exist. Also note that the single active core prevents multicore parallelism, reducing the incentives to add many cores. If this barrier is removed, SIHM processors could utilize homogeneous multicore interconnect networks to scale cores upward.

Design complexity

By using existing GP-CPU designs, SIHM does not require much hardware design effort. If custom cores were designed for the SIHM processor however, designing multiple unique cores for a single ISA would become a highly complex challenge.

SIHM processors use a common ISA, so no porting work is needed to target the multiple cores. The only additional software design challenges are to include OS support for picking and switching to the best core for a given workload.

Production cost

If cores from successive processor families are used, the additional area overhead of SIHM is low. Over time, technology scaling has given processor designers more transistors for the same silicon area. As a result, older core designs use much fewer transistors than modern designs. When fabricated at the same process technology, the area of older core designs is insignificant compared to modern designs. Additionally, sharing the L2 cache SRAM between all cores reduces the area overhead of adding additional cores.

Target market

Due to the support for a single general purpose ISA, SIHM's can target large computing markets. If the SIHM uses an already popular ISA, the SIHM can be used as a drop-in replacement to reduce power and energy consumption in existing systems without porting. This approach could quickly provide energy savings for mobile processors which have typically relied on several generations of ARM cores, and whose workloads do not require state of the art performance at all times.

1.2.6 Mobile SoC

Mobile phones typically rely on System-on-Chip (SoC) processors to efficiently handle workloads while maximizing battery life. These mobile SoCs contain a GP-CPU core (often designed by ARM) and a handful of ASIC-like accelerator cores. The accelerators are large and coarse grained, meaning that they implement large algorithm classes like 3D rendering, audio decoding, or radio basebands. The accelerators typically communicate with the GP-CPU and system memory DRAMs via a shared bus. SoCs with many accelerators and peripherals may bridge this bus to a secondary bus connecting UARTs and other low-bandwidth peripherals.

SoCs rely on DMA controllers (DMACs) to copy data between the system memory DRAM and accelerator SRAM I/O scratchpads, and many DMACs add dedicated channels to accelerators

to reduce shared bus contention. DMA is crucial for SoC performance by offloading time consuming memory transfers away from the GP-CPU.

Mobile SoCs are popular in many mobile phones and portable media players. TI's OMAP, Qualcomm's Snapdragon, and Samsung's line of SoCs are frequently used for these devices today.

Research designs have suggested relying on accelerators for frequently executed workloads and using a GP-CPU to perform less common operations when necessary. Under such an approach, the GP-CPU and unused accelerators can be turned off to save power, thus ensuring only specialized circuits are active in the common case. This approach is only feasible for domain-specific processors where "common workloads" can be identified before fabricating the processor. For example, a ULP processor utilized accelerators for wireless sensor networks to significantly reduce energy and power consumption [33] by utilizing performance improvements of up to 1000x, and scaling down voltage and frequency.

Execution improvement

As noted above, the dedicated accelerators used in mobile SoCs are known to improve performance up to 1000x. Of course, this performance improvement applies to the limited number of operations supported by the ASIC-like accelerators. These performance improvements can, and frequently are, traded off for reduced power consumption. In addition, accelerators frequently use dedicated voltage domains that can be independently turned off (VDD-gated) when not in use.

Customization

In contrast to the previously discussed accelerated architectures, mobile SoC accelerators are highly specialized for a specific algorithm. Furthermore, SoCs contain several unique accelerators, resulting in a much more heterogeneous architecture. The SoCs are unlikely to simultaneously compute all of the accelerated algorithms at the same time, so silicon area utilization at any given time will be much lower than homogeneous systems.

Mobile SoCs provide excellent power and performance improvements for the small subset of workloads targeted by its accelerators. These accelerators are typically too workload-specific to accelerate anything beyond this work set, and are inflexible to other domains.

Scalability

Mobile SoCs rely on a shared bus and a DMAC to transmit data, which are centralized and will not scale to large numbers of accelerators. The shared bus will quickly become a bottleneck, because every transfer from system DRAM memory to the accelerators or GP-CPU must go over the shared bus. If the number of accelerators grow, the traffic over the shared bus will grow with it and eventually saturate. The DMAC's central control and buffering of accelerator traffic will also become a choke point. DMACs often provide dedicated channels to high bandwidth accelerators, so that each accelerator can transfer data to system DRAMs without delay from shared bus contention. However, this centralized structure is clearly not scalable in its current implementation.

Design complexity

Unlike homogeneous multicore architectures, each accelerator core is unique. As a result, the accelerators cannot be replicated from a single design but must be implemented individually. Due to the typically small number of accelerator cores, the interconnect design complexity is low, making system integration of accelerators comparatively simpler.

Software must also be updated to efficiently make use of the accelerators. The operating system and shared libraries must be designed to manage memory transfers between accelerators, system DRAM memory, and the GP-CPU. Accelerators typically process workloads in batches to amortize data transfer costs. As a result, application software may need to adapt to a batched processing model, rather than individually working with small pieces of data.

Production cost

Mobile SoCs contain several accelerators in addition to the GP-CPU. As a result, the area consumed by the SoC increases with each accelerator. However, the most intensive work is performed efficiently by accelerators, reducing the need for a complex GP-CPU. As a result, a simpler GP-CPU can be used, reducing area increases brought on by the addition of accelerators.

Target market

The addition of coarse-grained accelerators limits mobile SoCs to the tasks which depend on popular, preset algorithms. These algorithms typically include support for audio, video, 3D rendering, radio baseband, and signal processing. As a result, these SoCs are well suited for mobile phones and media players, but few systems beyond portable media consoles.

1.2.7 Reconfigurable computing

Reconfigurable computing aims to map acceleration to all possible workloads. Rather than fabricating accelerators for specific workloads, reconfigurable processors contain field programmable gate array (FPGA) logic, which consists of many small look-up tables (LUTs) and D flip-flop (DFF) register memories. By configuring these components at runtime, FPGAs appear to implement different logic. The reconfigurable approach ensures that if it is possible to make an accelerator for an algorithm, an FPGA can implement it, no matter how obscure.

The downside to FPGA technologies is overheads: a 21x increase in area, 12x increase in power, and 3-4x decrease in performance [45]. Unlike ASIC approaches, which synthesize transistors and wires directly on silicon, FPGAs must map netlists into LUTs and DFFs. Although this intermediate mapping enables FPGAs to be flexible through reconfiguration, the mapping is also inefficient. This is not to say that FPGAs will always consume more area or use more power than a synthesized GP-CPU, because FPGAs may implement accelerators that improve performance and power by several magnitudes, more than compensating for FPGA overheads. However, custom ASICs implementing the same accelerator using the same process technology will always have smaller area and power costs, and better performance.

The FPGA business has long since matured, and many FPGA processors are commercially available from Xilinx, Altera, and several other companies. Most FPGA processors now contain a combination FPGA logic and ASIC-style non-reconfigurable cores, such as multipliers, small SRAMs, and GP-CPUs. This approach aims to mitigate the area, performance, and power overheads for these commonly used components, while simultaneously providing FPGA circuitry for rarer logic.

Execution improvement

FPGAs can provide the same ASIC-style accelerator performance improvements. Due to the performance and power overheads of the inefficient LUT and DFF substrate, the performance and/or power improvements is substantially degraded. Therefore, FPGAs are ideal for less common logic which would be economically unfeasible to fabricate as an ASIC, but a poor implementation style for commonly used logic such as GP-CPUs and commonly used accelerators.

Customization

FPGAs are uniquely homogeneous *and* highly customizable. From a fabrication perspective, FPGAs mostly consist of a highly homogeneous LUT and DFF fabric. From a circuit designer's perspective, FPGAs have the potential for unprecedented customization, since FPGAs can implement any arcane netlist. Further, the same FPGA can then be reconfigured for a different netlist later, as workloads change. As a result, FPGAs are uniquely flexible *and* highly customized. Also, the FPGA may be fully utilized unlike other highly customized architectures, because the entire FPGA fabric can be reconfigured to efficiently target changing workloads.

Scalability

Due to the regular nature of the FPGA fabric, FPGAs are highly scalable. However, building on-chip networks using the LUT and DFF building blocks is inefficient. This may motivate optimized, non-reconfigurable network implementations to improve large-FPGA efficiency.

Design complexity

Due to the separation of FPGA fabric design and the reconfigurable circuits implemented a layer above, FPGA design complexity is both simple and complex.

The FPGA fabric's design complexity is relatively low, consisting mainly of simple LUTs and DFFs. Although fabrics often contain non-reconfigurable multipliers, SRAMs, and simple GP-CPUs, designs for these components are relatively simple, and replicated throughout the FPGA fabric.

Circuit designs implemented on the FPGA fabric are decidedly more complex. Because

FPGAs make implementing custom hardware for arcane algorithms economically feasible, designing and integrating these custom accelerators introduces the complexities of mobile SoC processors to smaller design efforts. Adding "on-the-fly" reconfigurability (reconfiguring without processor downtime) adds a new level of complexity. Now, the processor must decide when and with which fabric regions to reconfigure. This is a unique and complex challenge for reconfigurable computing, since statically fabricated technologies cannot reconfigure.

Production cost

The production cost of FPGAs is comparatively cheap for low-volume designs, but high for high-volume designs. For low-volume designs, the monetary costs for one-time costs such as fabrication mask creation, time and cost overruns from design mistakes, and the long time to market makes ASIC fabrication prohibitive. FPGAs, although imposing a 21x area overhead, do not suffer from these overheads and make low-volume designs economically feasible.

However, the one-time overheads associated with ASIC fabrication are insignificant for high-volume designs, whereas the significant area overheads required by FPGA implementations are incurred for every chip. For this reason, the production cost of FPGAs is too large for high-volume projects.

Target market

Due to high flexibility, FPGAs can target virtually any low-volume market, since one FPGA processor design can be reconfigured for virtually any application domain. Due to production costs, FPGAs cannot compete on price for high-volume markets, which can afford to give up some processing flexibility given a large enough market for specific accelerated application domains.

1.3 Challenges

There are many accelerated architectures, each with a distinct approach to improve performance, energy, power, or all three. Each design has trade-offs in terms of the magnitude of execution improvements, future scalability, support for workload-specific customization, design complexity, product cost, and market potential. There is no "correct" accelerated architecture, rather, archi-

tectures better suited for certain expectations. However, it may be possible to combine approaches to maximize benefits.

Reconfigurable logic provides a unique combination of flexibility and specialization that can accelerate a wide swath of workloads without high-volume demand. However, FPGAs are unable to deliver the ultimate performance, power, and energy improvements found in the ASIC-style accelerators of mobile SoCs. A "many-accelerator" architecture, where each of the many ASIC-style accelerators targets a common workload, combines ASIC execution improvement and FPGA wide workload coverage.

In the many-accelerator vision, many accelerators for common algorithms are available on-chip. Applications will therefore benefit from ASIC-level acceleration for most of their workloads, without incurring FPGA overheads. Of course, not all algorithms will be common or mature enough to warrant fabricated accelerators. In these cases, a portion of the many-accelerator processor may contain FPGA fabric. And for algorithms that do not benefit from hardware acceleration, such as state machines and system supervision, the many-accelerator system will include a GP-CPU. However, the majority of workloads will be processed by non-reconfigurable accelerators to maximize performance, power, and energy characteristics, and will only use FPGA or GP-CPU logic for the rare instances that the non-reconfigurable accelerators do not apply. As a result, the many-accelerator approach will support ASIC execution improvement and FPGA flexibility to changing workloads.

There are many opportunities for new innovations and accelerated architectures, all of which are critical to realizing the many-accelerator vision. In this section, areas of accelerated architecture design ripe for innovation are discussed.

1.3.1 Rapid accelerator development

Above all else, the many-accelerator architecture requires a large number of accelerators. Unlike software libraries that utilize high-level languages (HLLs) such as C or Java, hardware accelerators are typically developed in Verilog or VHDL RTL languages. As a result, accelerators are difficult to develop. Algorithms with free software implementations commonly command tens of thousands of dollars at a minimum as RTL designs. From a design perspective, implementing a many-accelerator architecture would be prohibitively costly in terms of money or time via today's approach.

However, support for HLLs is emerging in the accelerator world. Some approaches produce RTL designs from existing languages, such as C/C++ [15]. Other HLL approaches use new languages created to represent the unique design constructs of hardware design, such as Bluespec [9]. With either approach, the time to design accelerators is greatly reduced. Although the HLL compiler may not produce the most efficient hardware designs, HLLs can enable larger design projects and wider design space explorations that would not be tractable using RTL's limited expressiveness. Extensions on the HLL approach have automated compilation tools to automatically identify and synthesize accelerators and co-design software machine code and accelerator hardware [72, 39].

HLL hardware compilers may one day make developing hardware accelerators as easy as designing software. Further, designing hardware and software may cease to be different processes.

1.3.2 Identifying algorithms to accelerate

Before implementing an accelerated system, it is first necessary to identify which algorithms to accelerate. First, identifying "common algorithms" is a subjective task and will require a significant amount of workload profiling. Initially, profiling calls to system functions and shared libraries can reveal which explicitly defined algorithms are the most popular and worth accelerating. As a long term approach, similarity analysis of source code repositories would reveal other algorithms that are frequently used in software, enabling additional common accelerators.

1.3.3 Minimizing accelerator area overheads

The additional area incurred by adding many accelerators must be kept to a minimum. To keep accelerator area low, the granularity of accelerators must change. Coarse-grained ASIC-style accelerators found in mobile SoCs that target entire workloads, such as video and audio decoding, work well when used independently. However, these accelerators introduce redundant logic when used in the same system, since many workloads use the same common algorithms. For example, both H.264 video and MP3 audio decoders utilize Huffman decoders and IDCTs. If these coarse-grained accelerators were included in the same processor, at least two copies of the Huffman decoder and IDCTs would be fabricated. Instead, medium-grained accelerators could reduce redundant logic, while preserving the ASIC-style performance improvements not found in fine-grained instruction level accelerators. In the medium-grain domain, accelerators would accelerate individual stages of

workload processing, such as the Huffman decoder or IDCT. These medium-grained accelerators could then be chained together to implement the same algorithms as coarse-grained accelerators, such as video or audio decoding. As a result, each medium-grained accelerator could be used for multiple algorithms, thus making each medium-grained accelerator more workload-flexible compared to coarse-grained accelerators. Also, duplicate logic area can be eliminated without incurring the performance or energy costs of shoehorning acceleration into a restrictive ISA.

1.3.4 Scaling accelerator systems

To realize the many-accelerator vision, processors will include many more accelerators than current mobile SoCs. This increased accelerator count is particularly inevitable if coarse-grained accelerators are divided into several medium-grain accelerators. However, current mobile SoC interconnects that utilize shared memory for inter-core communication, centralized arbiters such as DMA controllers, and poorly scaling interconnects such as shared buses cannot support an order of magnitude increase in accelerators.

Although tempting, existing scalable interconnect networks for homogeneous multicore architectures will not effectively map to the world of heterogeneous accelerator systems since many assumptions no longer hold. Because each accelerator core is unique, most cores will be idle or turned off. As a result, a fraction of accelerators will utilize the on-chip network. This would be highly unusual in a homogeneous multicore processor, where each core is typically active and interacting with the on-chip network in a regular manner. Further, accelerators tend to send and receive larger datasets than the messages passed over homogeneous multicore on-chip networks. A network designed for larger batches of data would be more efficient than the packet-switched networks favored in homogeneous multicore on-chip networks.

Networks designed for systems where a small percentage of cores are active and producing bursty batch transfers would provide efficiency gains for the many-accelerator architecture. In addition, a distributed memory and memory management architecture will be necessary to avoid the bottlenecks of a single system DRAM memory and centralized DMA controller management.

The work presented in this thesis pursues the implementation of an efficient architecture for many-accelerator systems. This architecture must efficiently satisfy the needs of single-algorithm accelerators, and scale to support tens or hundred in the same system. To design such a system, un-

derstanding the unique behavior of accelerators, rather than general purpose CPUs, is paramount. With this in mind, the following chapter discusses the development of an accelerator for maintaining compressed Bloom filters in a wireless sensor network.

Chapter 2

Accelerator composition

To understand the attributes and behavior of single-algorithm accelerators, this chapter details the development of one such accelerator. Although each accelerator derives much of its performance and energy improvements by specializing for different algorithms, the process to develop accelerators is not specific to any algorithm. By examining the development of a compressed Bloom filter accelerator for wireless sensor networks, accelerator characteristics useful for designing a many-accelerator architecture can be extrapolated. In particular, this chapter determines that much of the accelerator's silicon area is consumed by generic SRAM memory, not logic specific to the algorithm. This results in two takeaways: that a many-accelerator architecture should further investigate SRAM use in accelerators for optimization, and that the interconnect currently used to interface accelerators as part of a larger system are lacking.

2.1 Accelerator design requirements

Battery-powered embedded systems carefully manage energy consumption to maximize system lifetime. Wireless sensor networks (WSNs), made up of many "mote" devices, are often designed to operate for months without intervention. Sensor networks are typically used to monitor an environment and may be deployed in remote and hazardous locations. WSNs can consist of a hundred motes or more, and cover wide areas. As a result, mote software and hardware must consider energy consumption at every level.

Motes are simple, pocket-sized computers. Each mote contains a small battery that powers a

radio for wireless networking, a limited amount of memory, and a constrained processor. Aggregation, a widely researched field for reducing data transmissions by combining data on motes, reduces energy use by spending additional energy on computation to save a greater amount of energy on the power-hungry radio [51]. Increasing on-mote processing complexity will require additional computational hardware, demanding more energy. As sensor networks grow and generate larger data sets, these energy costs will continue rising.

Unlike PCs, embedded systems often execute a limited set of applications and have less need for general purpose functionality. Simple bit manipulations poorly utilize a general purpose processor. Complex operations, such as multiplication, require several cycles on a general purpose processor. Many embedded applications require support for these simple and complex operations and most existing systems must poorly utilize a general purpose microcontroller. In contrast, hardware accelerators tailor hardware to the application.

The presented in this chapter accelerator implements several operations for compressed Bloom filters, a data structure for efficiently storing set membership. These operations include support for inserting items, compression and decompression, and querying. Significant performance, power, and energy results over a general-purpose hardware solution for each custom operation is demonstrated. In addition, the benefits of hardware acceleration are explored in the context of three WSN applications: mote health monitoring, object tracking, and duplicate packet removal. For these benchmarks, Bloom filters improve network reliability and reduce radio transmissions by up to 70%. The hardware accelerator implementation provides significant gains in network latency (59%), computational delay (85-88%), and computational energy (98%) compared to executing the Bloom filter code on general purpose microcontrollers. Given these improvements, the benefits of architecting processors for hardware accelerators is shown. The Bloom filter accelerator can be VDD-gated when not in use so that it only uses energy when it will reduce total system energy consumption by an even greater amount.

The remainder of this chapter is organized as follows. Section 2.2 discusses related approaches to increase energy-efficiency and motivates the accelerator paradigm. Section 2.3 describes the algorithms needed for the Bloom filter hardware accelerator, and Section 2.4 discusses the architectural blocks needed to implement the approach. Power and performance advantages for the Bloom filter accelerator are quantified for specific operations (Section 2.5) and larger applications (Section 2.6).

2.2 Power and performance optimization strategies

Parallel processing is well known for increasing energy efficiency in general purpose computing. Designers distribute computation across several low-power cores rather than a single high-power core [57]. The low-power cores operate at a lower frequency, reducing voltage and power requirements. Several cores can be combined in one processor to meet computational goals. Assuming the lowest possible voltage is used, dynamic power is roughly proportional to nf^3, where n is the number of cores and f is the operating frequency; potential processing capacity is proportional to nf.

Ideally, power demands are minimized when many low-frequency cores are used. However, several factors limit the power reduction:

- Threshold voltage places a lower bound on voltage scaling. Subthreshold operation is possible but adds significant design challenges [74].
- Leakage current increases the power consumption of each additional core.
- Interconnect logic for communication between cores and shared memory requires additional power and may introduce bottlenecks.
- Software must be parallelized to run on all cores simultaneously

Hardware acceleration provides an alternative approach that may be more appropriate for embedded systems due to the more specialized nature of the workloads. Motes do not require the same amount of general purpose functionality as a conventional computing system and can be customized for better performance and lower power. Furthermore, acceleration leverages the same power-saving properties of parallelism by operating at low voltage-frequency combinations with high performance, while also capturing the benefits of explicit hardware support for simple operations such as bit-manipulations that are inefficient on general purpose cores. Additionally, accelerated systems do not require core interconnect logic or the software challenges of parallelized code.

Table 2.1: Bloom filter configurations

All configuration use a 16KB bit array and 32-bit elements. Bits per item applies to full Bloom filters

Config.	*Item capacity*	*Bits per item*	*Hash functions (k)*	*False positive rate*
1	13500	9.71	7	$< 1\%$
2	9000	14.56	10	$< 0.1\%$
3	6500	20.16	14	$< 0.01\%$

2.3 Bloom Filter Algorithms

Bloom filters provide a useful case study for an exploration of hardware acceleration. Using Bloom filters, many WSN applications can easily aggregate information and reduce the size of large data sets containing unique identifiers. These factors can reduce costly radio transmissions and lower overall mote energy usage. However, some Bloom filter operations may require several seconds of compute time on general purpose mote hardware, limiting the applicability of the approach and incurring high energy usage. By implementing hardware support for Bloom filters, WSN applications can achieve significant energy reductions without sluggish performance. The Bloom filter hardware accelerator improves performance and energy use by optimizing several algorithms in hardware. The accelerator natively supports Bloom filters, multiply and shift hashing, and Golomb-Rice coding support for data aggregation, near-random hashing, and data compression, respectively. The following sections describe these algorithms in detail.

2.3.1 Bloom filters

Bloom filters efficiently store set membership of large items by combining data in a large bit array. Using a small number of hash functions, $h_1 \ldots h_k$, Bloom filters reduce storage costs up to 70% [12]. Many applications, including spell checkers and distributed web caches currently use Bloom filters. Other work has also suggested the use of Bloom filters in hardware [62, 59, 22]. However, these works use Bloom filters for internal processor or network management and do not expose Bloom filters to applications. Section 2.6 explores several Bloom filter applications for wireless sensor networks.

The hardware accelerator implements a specific range of Bloom filter configurations: the bit

array is 16KB, up to 16 hash functions are available, and 32-bit items are supported. Initially, the accelerator sets every bit in the array to 0, to create an empty Bloom filter. It inserts items by hashing the item x_i with every hash function $h_1 \dots h_k$. The results of these hash functions $h_1(x_i) \dots h_k(x_i)$ are addresses to bits in the array, which are set to 1. As the accelerator inserts more items, the number of 1's in the Bloom filter increases. When inserting items, some bits may already be 1 due to previous item insertions writing to the same bit address.

Querying to check if an item x_i is in the Bloom filter is similar to insertion. The accelerator hashes the item with every hash function $h_1 \dots h_k$ and checks each bit's value at addresses $h_1(x_i) \dots h_k(x_i)$. If any hash function points to a 0 bit, the accelerator knows with certainty the item is not in the Bloom filter. The item is in the Bloom filter with high probability if all hash functions point to 1 bits, but cannot be known with certainty. These "false positive" errors, although rare, occur when other inserted items hash to the same bits as the queried item. The false positive rate can be pre-configured as required by the application, typically from 1% to 0.01% at the cost of item capacity.

Items cannot be removed from a Bloom filter. Hypothetically, an item could be removed by setting any of the item's corresponding array bits to 0. However, many inserted items may hash to the same bit, and removing one item may inadvertently remove several other items. All elements can be cleared by setting all bits in the array to 0.

The false positive rate, item capacity, and energy requirements to insert or query an item are determined by k, the number of hashes used by the Bloom filter. When k is larger, the false positive rate decreases. However, smaller values of k result in Bloom filters with a larger item capacity and lower energy cost per item insertion or query. This trade-off is illustrated in Table 2.1. A detailed analysis of Bloom filter configuration is available in [12].

Bloom filters merge by bitwise ORing bit arrays, assuming both Bloom filters use the same bit array lengths and hash functions. This property makes aggregating data in a WSN spanning tree a trivial task: parents can merge Bloom filters from child motes quickly, insert their own items, and transmit the aggregate Bloom filter to its own parent.

The Bloom filter is considered full when half of the array's bits are 1. At this point, further insertions will dramatically increase the false positive rate. Bloom filter storage is most efficient when full, as the bit array is always a constant length. For example, configuration 1 in Table 2.1

can store 32-bit elements using less than 10 bits when full.

2.3.2 Multiply and Shift Hashing

Multiply and shift hashing, described by Dietzfelbinger et al. [23], is simple, yet effective. Each hash function $h_1 \ldots h_k$ requires a hash key $HashKey_1 \ldots HashKey_k$. Hash keys are odd integers randomly chosen before the Bloom filter is used. The accelerator represents hash keys as 32-bit integers.

The hash h_i of element x_j is calculated using:

$$h_i(x_j) = \frac{(HashKey_i \times x_j) \bmod 2^{32}}{2^{32-b}} \tag{2.1}$$

where b is the number of bits in the Bloom filter bit array address. For the 16KB bit array used by the accelerator, $b = 17$. The modulo and divide are powers of two and can be efficiently implemented with a bit mask and shift.

2.3.3 Golomb-Rice Coding

The accelerator implements Golomb-Rice coding, a popular compression and decompression method used in Apple's Lossless Audio Codec (ALAC) and Lossless JPEG (JPEG-LS) [54, 63]. As noted in Section 2.3.1, a Bloom filter contains more 0s than 1s until filled. Therefore, sparsely filled Bloom filters (under 70% full) can reduce Bloom filter size through Golomb-Rice coding. The algorithm, a form of run length encoding, is simple to implement, and therefore power efficient.

The number of 1s in the bit array are first counted to determine the "remainder part" length l. The relation between 1s in the bit array and l is precomputed; only a quick lookup is needed to determine the remainder part length.

Second, the bit array is iterated from start to finish, scanning for run lengths of 0s between 1s. For each run length of n 0s, the remainder part $r = \lfloor \frac{n}{2^l} \rfloor$ and quotient part $q = n \bmod 2^l$ must be calculated.

After calculating r and q, the accelerator writes r 0s to the compressed bit stream, followed by a 1. q is then written directly, using l bits. This process is used to write all run lengths in

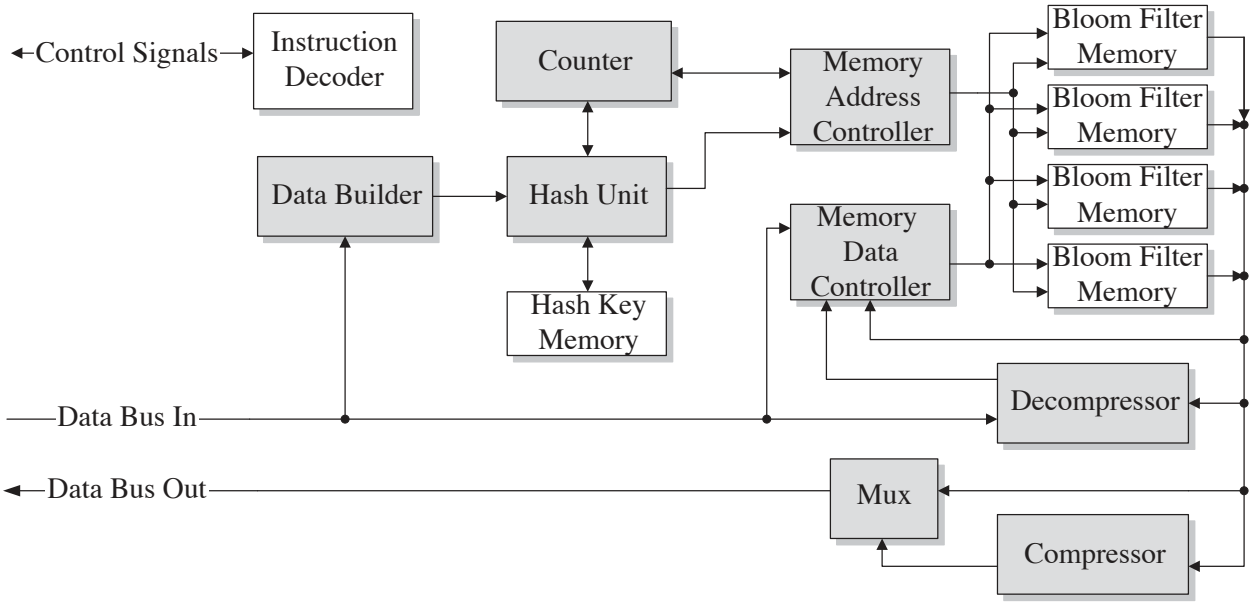


Figure 2.1: Bloom filter hardware accelerator hardware flow
Arrows indicate the direction of information, shaded blocks indicate modules controlled by the Instruction Decoder.

the uncompressed bit stream until the end is reached. The second step's implementation does not require any expensive divisions or modulos; a counter is kept of the current 0 run length. If the next bit is a 0, the counter is incremented. If the counter reaches 2^l, a 0 is written to the compressed stream and the counter is reset. If the next bit is a 1, a 1 is written to the compressed stream, followed by the counter's value using l bits. Therefore, Golomb-Rice compression can be reduced to many simple bit operations.

2.4 Accelerator Architecture

The Bloom filter accelerator leverages the mote architecture described by Hempstead, et al. [34] which provides a framework for custom hardware accelerators. The architecture proposes a lightweight event processor for managing power and offloading tasks to hardware accelerators. High-level events and tasks are decoded on the event processor and deployed to accelerators via memory mapped operations. A simple processor executes any operations not explicitly handled by accelerators. Hardware accelerators are synthesized with standard cells (e.g. ASIC flow) or through a shared on-chip programmable FPGA substrate.

The Bloom filter accelerator supports a 16-bit data bus and consists of several modules, illustrated in Figure 2.1. In the following sections, each major module in the Bloom filter accelerator is examined, and design decisions for reducing energy and delay is discussed.

2.4.1 Bloom Filter Memory

The Bloom filter bit array is stored in four 2K x 16-bit SRAM memory modules. The bit array is stored sequentially by address, so that bits are stored in the following order: Module1[0], Module2[0], Module3[0], Module4[0], Module1[1], and so on. A four-module configuration was selected to provide access to all four memory modules simultaneously, boosting performance by up to 4x. Each SRAM can access one address per cycle, so increasing the number of SRAMs available at a given cycle can greatly improve performance. Additionally, the accelerator uses four modules with a 16-bit data bus rather than one module with a 64-bit data bus because some Bloom filter operations only use one block per cycle. In this case, the unused three blocks can be disabled to reduce dynamic power consumption. The design does not use a larger data bus because significant additional logic would require more power and area, and wider bus lengths would rarely be fully utilized and provide less corresponding performance improvements.

2.4.2 Memory Data Controller

The Memory Data Controller manages data stored in the Bloom filter memory. The accelerator supports several Bloom filter operations, each writing to memory in a distinct style. Insertions and queries only modify one bit at a time, while other operations may modify one block or four blocks per cycle. The Memory Data Controller is responsible for ensuring each operation can write as many or as few bits as is required.

The Memory Data Controller also counts the number of 1s inserted into the Bloom Filter at every cycle. The accelerator uses this counter during compression operations to eliminate the need for an additional full memory iteration as described in Section 2.3.3. As previously noted, memory access can be a bottleneck, so this optimization is critical for performance.

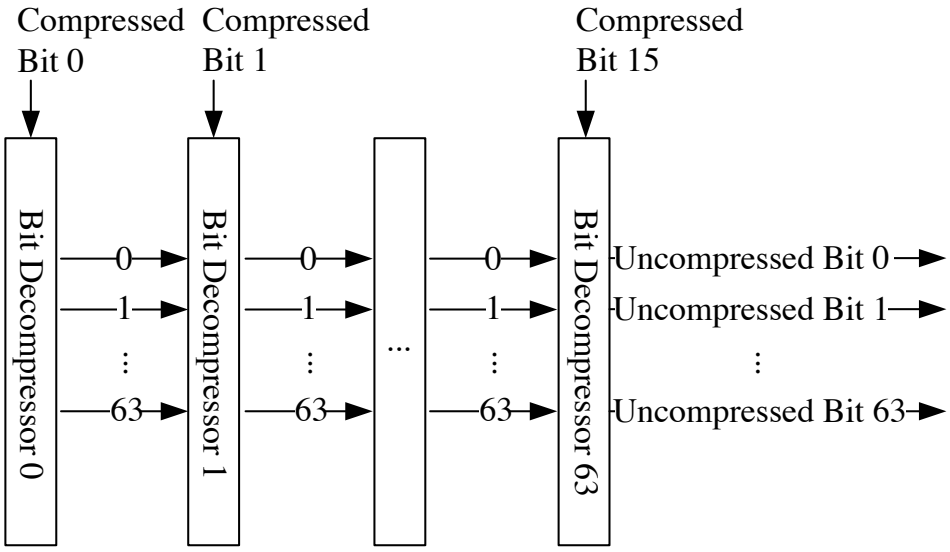


Figure 2.2: Decompressor information flow

2.4.3 Memory Address Controller

The Memory Address Controller coordinates with other blocks to correctly set the addresses of each of the four Bloom filter blocks. Although item insertion and query operations randomly jump from bit to bit in memory, some operations may sequentially read one module at a time, and others read sequentially from all blocks simultaneously. During sequential operations, the Memory Address Controller remembers where processing ended in the last cycle so that the operation can be easily resumed.

2.4.4 Decompressor

The Decompressor reads 16-bit Golomb-Rice encoded Bloom filter blocks from the data bus and unpacks up to 64 bits of uncompressed Bloom filter. The Decompressor guarantees the entire compressed block will be processed, or 64 bits of uncompressed Bloom filter will be unpacked. These derive from the 16-bit data bus and the 64-bit width of Bloom filter memory. Although these limits require significant additional logic, this higher performance design is supported to avoid elevated computation delays when processing Bloom filters containing many elements.

The Decompressor is composed of 16 serially-connected bit decompressors, illustrated in Figure 2.2. This design allows each compressed bit to be decompressed serially. Although each bit could be decompressed in parallel and reassembled, the serial design allows bit compressors to be

disabled when the uncompressed stream is full, thus reducing dynamic power. A dynamic style would increase the speed of decompression, but is unnecessary due to the slow 100 KHz clock frequency used by the Hempstead processor.

2.4.5 Compressor

The Compressor design is similar to the decompressor design and is composed of 64 serially connected single-bit compressors. The Compressor reads 64 bits of uncompressed data from the Bloom filter bit array, producing up to 16 bits of compressed data per cycle. These bit limitations are due to memory access and data bus limitations respectively. As a result, compressed Bloom filters can be produced 4x faster than uncompressed Bloom filters. Supporting these guarantees requires additional logic, but gains in performance make this addition worthwhile.

2.5 Accelerator Evaluation

This section evaluates the design decisions discussed in Section 2.4 by comparing power, energy, and performance of the hardware accelerated design against a general purpose hardware design paradigm.

The Bloom filter hardware accelerator, with the exception of Bloom filter memory, was implemented in Verilog and synthesized for a commercial 130nm process using Synopsis Design Compiler, Encounter, and Cadence. The accelerator area is $792,850 \mu m^2$ and uses $1.217M$ transistors. Power calculations are based on Design Compiler estimates, using the `check_power` command on high effort. Bloom filter memory was generated by the Faraday Memaker tool and power estimates were profiled using Synopsis HSIM simulations. The accelerator implements two-phase clocking, operates at 1.2V and supports a 100 KHz clock frequency.

As the placed and routed design shows in Figure 2.3, SRAMs (shown as black boxes) consume 79% of the accelerator's area, and 21% is used by accelerator logic (below and to the left of SRAMs). This result runs contrary to what many might assume. Although hardware acceleration depends on customization to achieve power and performance improvements, generic SRAMs rather than specialized logic consumes the majority of the hardware accelerator's area.

Figure 2.4 shows the distribution of power between the larger elements of the accelerator.

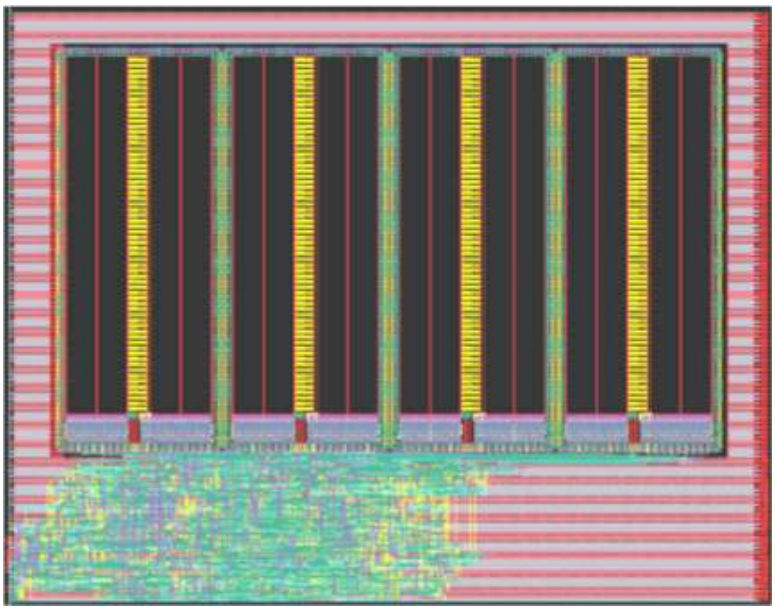


Figure 2.3: Placed-and-routed Bloom filter accelerator design
SRAMs (79% of used area) are shown as black boxes, logic (21% of used area) is located in bottom left corner.

When synthesized individually, each module requires roughly 10% more energy than shown, however, Design Compiler is able to reuse logic between modules to reduce total system energy. Therefore, this analysis assumes shared logic savings is proportional to the total energy cost of each module, and scales each module's power equally to match Design Compiler's system estimate. Again, generic SRAM memory consumes a large portion (45%) of the accelerator's power budget.

The general purpose design uses the Bloom filter software implementation for motes in Chang, et al. [13]. This software implements Bloom filter operations exactly as described in Section 2.3. The software was written in nesC for TinyOS 1.1.15 [36] and tested directly on the TMote Sky mote. The TMote Sky features a relatively powerful 16-bit, 8 MHz TI MSP 430 processor with 10KB of memory. The analysis compares the hardware accelerator with the MSP 430 processor due to its wide popularity in the sensor network community.

Due to memory limitations of the TMote Sky, only 8KB of memory is available for the Bloom filter bit array in the general purpose implementation. Since the accelerated implementation uses a 16KB Bloom filter, the accelerator will use additional power supporting the additional memory, as well as extra cycles to work on a Bloom filter twice as large. Yet, the accelerator logic demonstrates

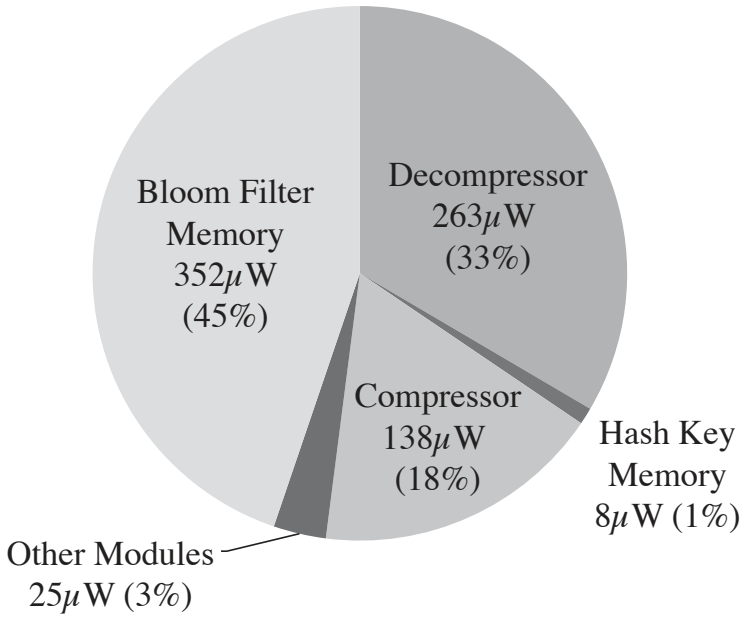


Figure 2.4: Average power usage of Bloom filter hardware accelerator modules

significant performance and energy savings despite this handicap.

Timing figures for the accelerated implementation are generated by counting the number of cycles used. Total system power for the accelerator design is 886μW, of which 786μW is expended by the Bloom filter accelerator. The remaining 100μW is consumed by the Hempstead event processor and other infrastructure logic.

Timing figures for the general purpose implementation are obtained through experimentation. Operations are executed on the TMote Sky and timed using internal microsecond and millisecond clocks for high accuracy. Using an average TMote power of 4.86mW, derived from the TMote Sky datasheet [19]. Therefore, the general purpose implementation's processor power requirements are almost 450% higher than the accelerated implementation.

For both implementations, energy is calculated as $Power_{avg} \times Time$.

2.5.1 Item Insertion and Querying

Item insertion is highly efficient in accelerator logic due to native support for the complex multiply and shift hashing operation. Accelerator logic requires significantly less time for insertions in all cases, as illustrated in Figure 2.5. Furthermore, insertion uses 97% less energy per insertion than

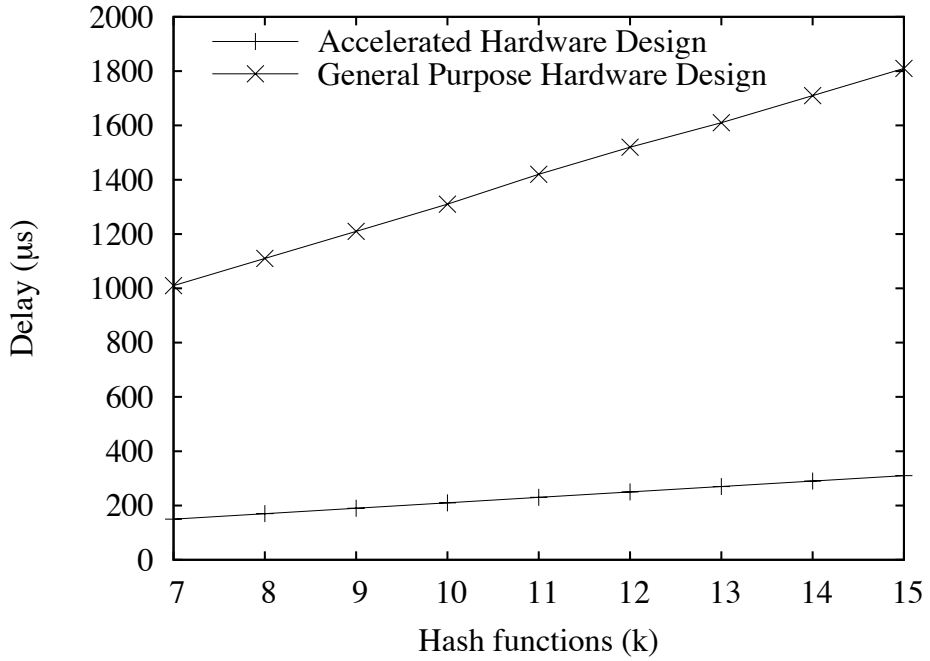


Figure 2.5: Item insertion times of application-specific hardware design logic and general purpose design logic

general purpose logic, regardless of the number of hash functions used.

Several factors contribute to the accelerator's advantage. First, the accelerator logic can hash during one cycle due to its native support for multiply and shift hashing. On the contrary, the MSP 430 processor used in the general purpose implementation only supports 16-bit math and must spend many cycles to complete the multiplication operation.

Further, the accelerated implementation does not require additional logic to perform the required bit shift. Instead, the accelerator's logic simply uses bits 31 through bits 15 (bit 0 is the least significant bit). The MSP 430 does not natively support this bit selection and must spend additional cycles on a 32-bit shift to obtain the corresponding bit address.

Memory operations limit the speed of as the accelerated implementation, as only one memory read or write can be performed per cycle. Therefore, the accelerator's insertion implementation requires 2 cycles per hash (one to read the block, another to write the modified block back). Parallelizing item insertion would require significant additional logic due to the seemingly random bit addressing caused by the hash function. Although support for processing up to four hashes could theoretically be possible due to the four Bloom filter memory modules, the block location

of each hash is unknown until calculated. Furthermore, all hashes could theoretically point to the same block, making simultaneous bit insertions impossible.

Querying items contained in the Bloom filter is similar to item insertion: the item is hashed k times and the bit at address $h_i(x_j)$ is verified to be 1. This process takes roughly half the time of insertion on accelerated logic because only one memory read is required for each hash. The general purpose implementation is time-bound by the hash function, however, so performance is largely equal to insertion.

2.5.2 Compressing Bloom Filters

The accelerator improves Bloom filter compression performance up to 1800%, as shown in Figure 2.6, and reduces energy consumption up to 99%. The key to these accelerator-based improvements is custom support for Golomb-Rice coding. When implemented in software for general purpose systems, each uncompressed bit must be examined to count run lengths of 0 bits. When a 1 is encountered, another lengthy set of bit operations must be performed to determine the correct sequence of compressed bits. As more elements are inserted into the filter and the frequency of 1s increases, the quantity of run lengths grow and additional work is required to compress. In general purpose software, this additional work increases the compression delay. The accelerator requires additional cycles as well, but provides a fast upper bound on compression delay. The accelerator's compressor guarantees a compressed 16-bit block will be produced or 64-bits of uncompressed data will be processed every cycle. Therefore, compression can never exceed 81.92ms and is often faster.

As noted in Section 2.3.3, the number of 1s in the bit array must be counted to determine l, before compressing run lengths. Memory access is slow in the accelerator implementation due to the 100 KHz clock frequency, and iterating through the memory would require an additional 20.48ms. To avoid this penalty, the accelerated implementation counts the number of ones in the filter as they are inserted. Therefore, the accelerator requires only one memory pass. The bit tracking technique is not used in the general purpose implementation due to lack of hardware support. Adding support in software would require several cycles per insertion or query, operations frequently used in many applications. Adding bit tracking support for lengthy merge operations would also require significantly larger delays. Performing two passes of memory, a fast process in the 8 MHz general purpose implementation, requires less delay overall in the general purpose

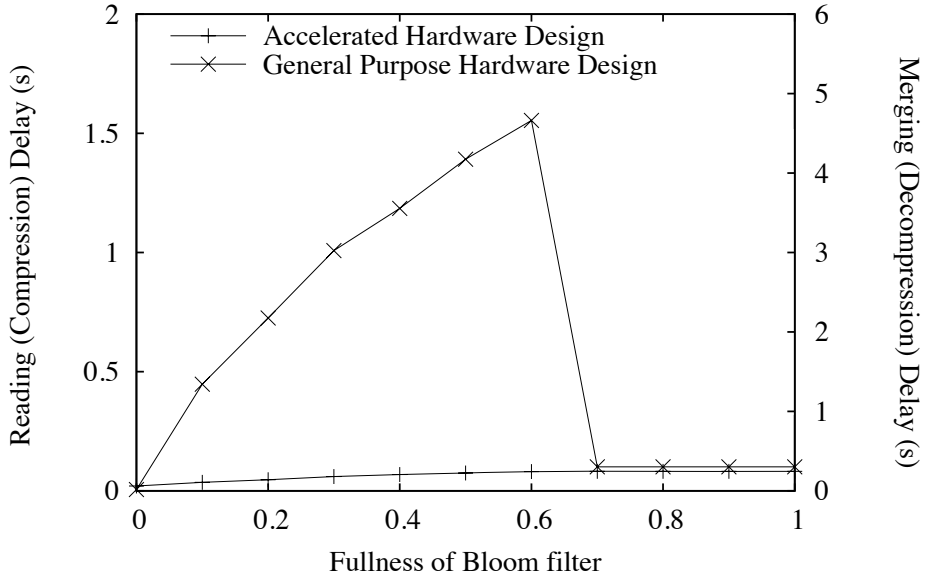


Figure 2.6: Bloom filter reading and merging delay at a 1% false positive rate
The accelerated implementation uses a 16KB Bloom filter; the general purpose implementation uses an 8KB Bloom filter. Uncompressed Bloom filters are used after reaching 70% fullness.

system.

At 70% of capacity, run lengths become too small to effectively compress, and Bloom filters are delivered uncompressed. The general purpose implementation no longer compresses and simply reads from memory. Although the accelerated implementation is only 19% faster when the Bloom filter is approaching capacity, recall that the accelerator's Bloom filter is twice the size of the general purpose implementation. If both implementations used equivalently sized bit arrays, the accelerated implementation would perform 59% faster. Further, the accelerator could reduce the uncompressed Bloom filter read delay by 75% if a wider data bus is used in a future architecture.

2.5.3 Merging Compressed Bloom Filters

Bloom filter merging uses bitwise ORs to combine a foreign Bloom filter with the filter stored in the bit array. The foreign Bloom filter, delivered over the data bus, is processed over several segments. If compressed, each foreign Bloom filter segment must first be decompressed. Meanwhile, the corresponding Bloom filter segment stored in memory is loaded. The two segments are bitwise ORed and saved back into the bit array memory.

Bloom filter merging performance resembles Bloom filter compression performance, as shown in Figure 2.6. The accelerator performs up to 2700% faster and can reduce the energy cost by more than 99%. This performance boost is largely due to the accelerator's decompressor design. The decompressor guarantees the entire compressed Bloom filter segment will be decompressed, or four blocks of uncompressed memory will be processed. The first guarantee provides an upper bound of 163.85ms per merge, but the second speeds up the process when the foreign Bloom filter is highly compressed.

The large gains are only obtained when using compressed Bloom filters. Rice-Golomb coding is relatively inefficient in general purpose hardware due to the large number of bitwise operations. However, once uncompressed Bloom filters are used beyond 70% capacity, general purpose performance noticeably improves. The general purpose implementation appears to operate faster beyond 70% capacity due to the memory size disparity between implementations. If both implementations used the same bit array size, the accelerated implementation would require 34% less time and 88% less energy.

When low or no compression is used, the data bus limits performance. Because each Bloom filter segment requires two cycles (once to read from the bit array and once to write), 16,384 cycles are required to perform a merge in the worst case. If a larger data bus were used in a future architecture, merging delay could be reduced by an additional 75%.

2.6 Application Evaluations

In this section, several distinct Bloom filter-based wireless sensor network applications are examined to demonstrate Bloom filter gains and to quantify accelerator performance and energy improvements. Each application represents a different class of Bloom filter use. The mote status application shares a Bloom filter across a sensor network, so that a central server can check if any motes in a large network require attention. In the object tracking application, each mote in the network individually records the unique identifiers of sensed objects in a private Bloom filter, periodically transmitting the filter to a central server. The duplicate packet removal application uses a Bloom filter to locally store identifiers of each packet received to quickly remove any packets duplicated by routing errors.

Mote networks may contain thousands of motes in the future, and managing mote operation will be critical in maintaining reliability. Mote networks are typically routed in a spanning tree formation and support multi-hop routing. A central server will connect to the root mote to send, analyze, and store information from the sensor network. If a mote is not close enough to directly transmit data to a desired mote, data can hop across several intermediate motes to reach its destination. For example, if a mote wishes to send data to the central server, the mote would send a packet to its parent mote, which sends the packet to its own parent mote, eventually reaching the server by way of the root mote. Extremely large mote networks can easily become saturated if storage and transmissions are not properly managed.

The following examples assume the sensor network uses a two child per parent routing tree structure. The example also assumes mote radios have a 40kbps effective data rate (does not include transmission overhead) [14]. All calculations are estimations based on the accelerator analysis from Section 2.5 and on-mote timing profiling of the general purpose implementation software.

2.6.1 Mote Status

Spanning tree topology can be problematic for the root mote and motes nearby. The root mote must forward every packet sent from the sensor network to the central server; nearby motes must also handle large amounts of network traffic. In many cases, the root mote will not be able to forward data to the server quickly enough, resulting in dropped packets and poor quality of service. Even if the radio can handle the network traffic, the radio will quickly exhaust the root mote's energy and cut off the sensor network from the server.

Bloom filters can greatly reduce the transmission load on these taxed motes by efficiently aggregating mote status information, such as low battery warnings, within the network. In this example, a sensor network of 128,000 motes demonstrates this application's ability to monitor extremely large sensor networks by using Bloom filters. Each mote uses a corresponding 4-byte identifier and the example assumes 10% of the mote batteries are low at any given time. A mote can send a low battery alert by periodically transmitting its unique identifier (UID) to the server via its parent. Leaf motes on the boundaries should individually send these UIDs without Bloom filters. As Figure 2.7 indicates, Bloom filters are only efficient when at least 15% full, or when about 2000 items are inserted with a 1% false positive rate. As these low battery alerts hop from parent to

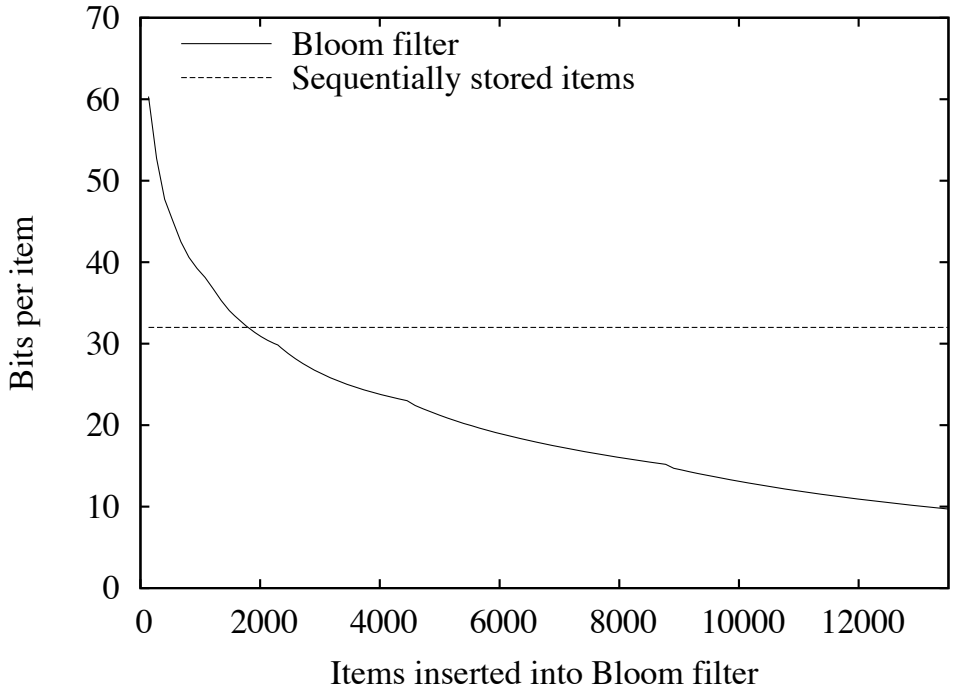


Figure 2.7: Storage cost per item for a 16KB Bloom filter and 1% false positive rate Bloom filters are more efficient than sequentially storing 32-bit items when the bit per item cost is under 32 bits.

parent, approaching the root mote, each parent mote will need to forward twice as many alerts as each child. When a mote has received 2000 or more items, it should create an empty Bloom filter and insert these elements. This Bloom filter will then be sent to the following parent mote, which will merge any Bloom filters it receives, insert UIDs sent sequentially without Bloom filters, and insert its own UID if its battery is low. This process of merging Bloom filters and inserting single UID items will continue as Bloom filters approach the root mote. The root mote will finally deliver a single Bloom filter, containing all low battery alerts, to the server. The server can then query the Bloom filter for each UID in the network to discover which motes require attention. Additionally, the server can track alerts over time to reduce errors from false positives: by identifying motes which consistently report low batteries, the server can remove erroneous alerts that sporadically appear. Meanwhile, all motes will clear their Bloom filters and restart the process as needed.

Both accelerated and general purpose approaches implement the same algorithms, so both are capable of reducing transmissions near the root mote up to 70%, thus reducing use of the mote's most power-hungry component. The accelerated implementation significantly reduces Bloom filter

end-to-end delay: the maximum delay from a mote issuing a low battery alert to the server detecting the alert is reduced from the general purpose implementation's 46s to 19s, assuming no transmission errors. Furthermore, the accelerated implementation reduces Bloom filter computation energy costs by 98% to 2.73mJ for every network-wide mote status scan.

2.6.2 Object Tracking

Previous work has used mote networks for object tracking [70]. Motes can identify objects by a unique ID using technologies such as RFID. This application uses Bloom filters with a sensor network to find packages in a busy package delivery warehouse. The example assumes each package has an RFID tag, so motes can detect package UIDs when nearby.

As packages move within the vicinity of a mote, the mote will wirelessly read the package's UID and store it in the mote's Bloom filter. When the Bloom filter becomes full, the mote will send the Bloom filter to the server. Note that the object tracking application does not merge Bloom filters with other motes. Instead, its Bloom filter is forwarded by other motes to the server for analysis. When a package is lost, the server looks at the most recently received Bloom filter for each mote and queries each to see if any have seen the package. If a false positive causes the package to appear in multiple places, previous Bloom filters can be examined to correctly identify the package location.

This merge-free approach requires additional latency: Bloom filters only store data for one mote, so more time is required to fill the Bloom filter. However, this technique also ensures Bloom filters are sent when they are full and store items most efficiently. When latency is not critical, individualizing Bloom filters can improve transmission energy costs at every hop, not just near the root.

To build each Bloom filter, motes must clear the Bloom filter, insert enough UIDs to fill the filter, and read the Bloom filter for transmission. With a false positive rate of 1%, the accelerator is able to reduce Bloom filter computation to 2.13s. This 85% reduction in delay over the general purpose software design corresponds to a 97% reduction in computation energy consumption.

2.6.3 Duplicate Packet Removal

Bloom filters are well equipped for removing duplicate packets [32]. Wireless sensor networks are particularly susceptible to duplicate packets due to wireless transmission errors. By using Bloom filters to track whether a packet was previously received, these transmission errors can be filtered out. When a mote receives a packet, the mote creates a unique packet identifier from the source packet's UID and the packet's sequence number. Motes query the Bloom filter using this packet identifier. If the packet is found, the mote processes the packet and sends an acknowledgment to the source mote to indicate that the packet was received. The mote also inserts the packet's identifier into the Bloom filter. If found, it likely received the packet and ignores it. However, the mote sends an acknowledgment to the source mote indicating a duplicate packet because false positives may cause the mote to mistakenly ignore an original packet. However, the source mote will realize the mistake upon receiving the acknowledgment and resend the same packet with a new sequence number. Dropped packets will be detected when no acknowledgment is received by the sending mote. In this case, the packet will be resent with the same sequence number.

Although this application does not transmit Bloom filters, the accelerator improves storage ability and reduces search time. The Bloom filter accelerator stores more than 200% additional packet UIDs and provides extremely fast search times. Individually stored packet identifiers would require significantly more physical memory and additional searching algorithms.

When working with frequent radio transmissions, delays must be minimized. In the worst case, each delivered packet requires one item query and one item insertion to eliminate duplicate packets. For a 1% false positive rate, this process requires 240μs with the accelerator. This performance boost corresponds to an 88% delay reduction and 98% computation energy reduction over the general purpose implementation.

2.7 Takeaways

During the development of the Bloom filter, two major observations were made: SRAMs consume the majority of accelerator area, and the interconnect used to interface accelerators limits the number and granularity of accelerators. With these concerns in mind, the following chapter develops an architecture to address these issues.

Chapter 3

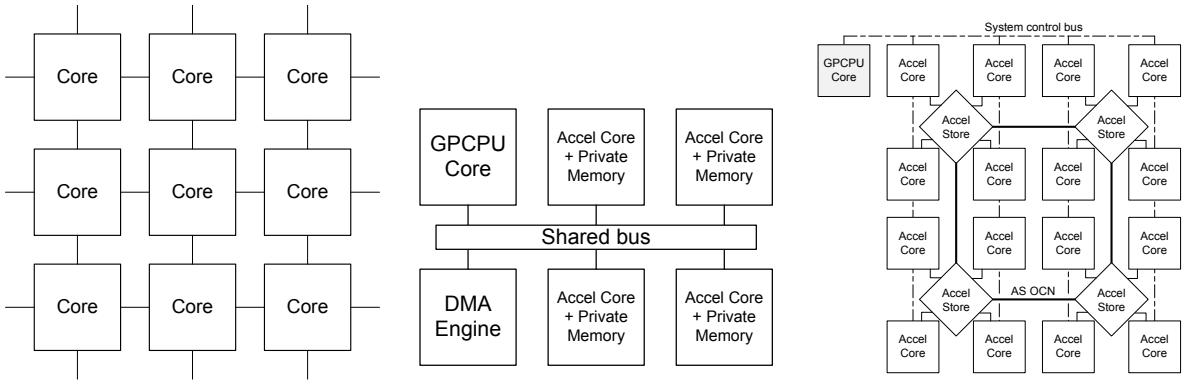
Accelerator store

As Chapter 1 discussed, the end of threshold voltage scaling has led architects to turn off cores to stay within power budgets. These *dark silicon* transistors are unused and therefore wasted. There is no reason to create more copies of the same core design than can be powered, but including a heterogeneous mix of core designs allows the processor to turn on the most efficient cores for current workloads, and only the least efficient cores will be dark at any given time.

Hardware accelerator cores, including the previously discussed Bloom filter accelerator, represent the frontier of customization-fueled performance: for maximum efficiency, each accelerator implements a single algorithm. By building processors containing many accelerators, not just identical copies of the same general purpose core, architects can again achieve frequency-clock scaling performance gains. The key is taking advantage of, rather than falling victim to, dark silicon using hardware accelerators.

In order for many-accelerator systems to be a viable performance-enhancing solution, it is important to first understand characteristics of several accelerators and develop a flexible framework that ties them together. This chapter presents the accelerator store (AS), a shared-memory framework, which allows for efficient allocation and utilization of resources in many-accelerator architectures.

To successfully design the AS and efficiently support many-accelerator systems, a deep understanding of the accelerators is necessary. Section 3.1 surveys eleven commercial and open source accelerators, revealing that generic SRAM memories consume 40 to 90% of the area in each accelerator, showing lessons learned from development of the Bloom filter accelerator were not unusual.



(a) Traditional multi-core (b) Traditional SoC (c) Many-accelerator

Figure 3.1: Comparison of architecture styles

This study then categorizes these private accelerator memories into four categories based on capacity, bandwidth requirements, and latency sensitivity. The analysis finds that in many cases large private SRAM memories embedded within accelerators also tend to have modest bandwidth and latency requirements. Sharing these SRAM memories between accelerators would reduce the amount of on-chip memory through amortization, thereby shrinking total processor area. This area reduction can be used for cost savings or to place additional accelerators for even greater diversity and performance improvements. A well-architected accelerator store along with careful selection of shared SRAM memories leads to very little overhead compared to private-memory based systems.

To efficiently share memory between accelerators and optimize many-accelerator systems, Section 3.2 presents the accelerator store's design. The accelerator store supports ways to combine multi-core and customized logic as illustrated in Figure 3.13:

- *Support for shared SRAM memories:* The AS allocates memory to accelerators on an as-needed basis.

- *Centralized model/decentralized scalability:* The AS presents a centralized view of shared accelerator memories to keep interactions with software and accelerators simple, but is physically distributed. A many-accelerator system can use multiple ASes, with a group of accelerators clustering around each AS as shown in Figure 3.1(c). The multi-AS design can scale to hundreds of accelerators.

- *Fine-grained accelerator support:* The AS's decentralized implementation supports accelera-

Table 3.1: Accelerators studied

Accelerator	*Function*	*Area used by SRAM memory*
AES	Data encryption	40.3%
JPEG	Image compression	53.3%
FFT	Signal processing	48.6%
Viterbi	Convolutional coding	55.6%
Ethernet	Networking	90.7%
USB (v2)	Peripheral bus	79.2%
TFT Controller	Graphics	65.9%
Reed Solomon Decoder	Block coding	84.3%
UMTS 3GPP Decoder	Turbo convolutional coding	89.2%
CAN	Automotive bus	70.0%
DVB FEC Encoder	Video broadcast error correction	81.7%
Average		69.0%

tors in greater numbers than the handful found in today's SoCs. Many-accelerator systems may not just add new accelerators, but also decompose accelerators into multiple fine-grained accelerators. These finer-grained accelerators implement more commonly used algorithms and are less application specific.

- *Flexible abstraction:* The accelerator store's refined accelerator and software interfaces minimize overheads and open many research avenues.

Section 3.3 evaluates an RTL implementation of the accelerator store in multiple systems, demonstrating that area reduces by 30% while maintaining customized logic's superior performance and energy, and that performance and energy overheads due to added memory latency and contention are minor. Section 3.4 presents related work.

3.1 Accelerator Characterization

To design an efficient many-accelerator architecture, accelerators that comprise many-accelerator systems must first be analyzed. The analysis begins with several commercial and open source accelerators, and finds that generic SRAM memories consume between 40% to 90% of each accelerator's area. SRAMs are the biggest consumer of accelerator area, but amortizing memory through sharing will greatly reduce this cost. Some accelerator memories are better shared than others,

so memory access patterns in four accelerators are characterized based on capacity, bandwidth, and sensitivity to memory latency. This analysis results in a simple methodology to select which accelerator memories to share and which memories should remain private.

3.1.1 Accelerator composition characterization

The analysis begins with the composition of several open source and commercially developed accelerators when synthesized for ASIC fabrication. These accelerators implement algorithms from several widely used domains, including security, media, networking, and graphics. It is impossible to directly obtain composition data with a standard ASIC toolchain because many accelerators utilized FPGA-specific logic blocks. Instead, each accelerator was synthesized using the Xilinx ISE 10.1 FPGA synthesis toolchain to obtain FPGA memory and logic statistics. Previously measured scaling factors [45] were then applied to obtain ASIC composition figures. The analysis shown in Table 3.1 demonstrates that SRAM memories consume an overwhelming amount of area in all accelerators, up to 90%.

Memory is therefore the best target for area optimization. Leveraging the large numbers of accelerators in the many-accelerator system creates more opportunities for area reductions by sharing memories between accelerators. At any given time, some accelerators will be actively processing and turned on, while most accelerators will be dormant and VDD-gated off. These off accelerators continue consuming ASIC chip area without providing any function to the system. Unlike private accelerator memories that must be provisioned at circuit design time, shared accelerator memory can be dynamically assigned to accelerators at runtime. Via sharing, memory can be effectively provisioned to accelerators when needed, without the power, performance, and cost overheads of reconfigurable logic. This approach reduces the area cost of accelerator memory from the sum of each accelerator's memories to the much lower sum of memories used by accelerators at any point in time.

Shared memory also creates new memory reductions by eliminating over-provisioning and by merging redundant memories. Accelerator designers do not always know how their accelerators will be used, and over-provision memory to add flexibility. For example, a 1024 point FFT may also support 256 and 512 point FFTs. Although the core computation between any of these FFTs is the same butterfly operation, the 1024 point memory requires 4x of the memory used by the 256

point FFT. In another light, using the over-provisioned 1024 point FFT to perform a 256 point FFT wastes 75% of the FFT's memory. Although a smaller FFT would be more efficient in this case, the processor may be used for other applications requiring a larger FFT or the application may switch between FFT configurations. Memory sharing eliminates over-provisioning by assigning memory as required by the accelerator at runtime.

Merging memories with memory sharing reduces area costs and redundant transfers. Most accelerators contain memories for storing input and output data. Outputs from one accelerator often become inputs for another accelerator, and output memory and input memory store the same data. Sharing can merge these two memories into one shared memory, reducing memory area in half. Memory merging also reduces data transfers because copies between the two unshared memories are no longer required.

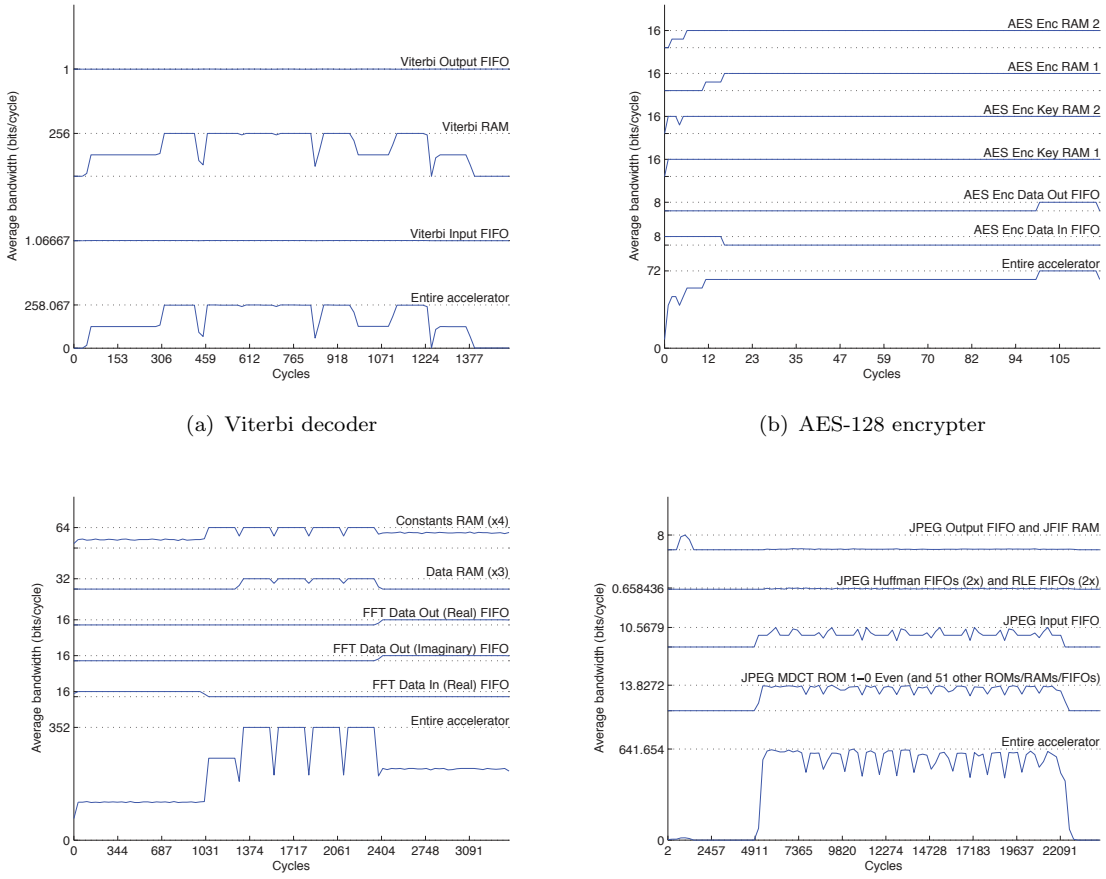
During early experiments, all private accelerator memories were initially shared. This indiscriminate approach saved a large amount of chip area but incurred significant performance overheads. A small portion of accelerator SRAM memories were not amenable to sharing due to extremely high bandwidth requirements or sensitivity to memory latency. To identify memories that can be shared effectively and memories that should remain private, this analysis characterizes SRAM memories from four accelerators, resulting in significant area reductions and minimal performance overheads.

3.1.2 Memory access pattern characterization

The access patterns of each memory was profiled using RTL for the first four accelerators (Viterbi, AES, FFT, and JPEG) and results shown in Table 3.1 to determine which attributes result in poorly performing shared memories. The function and design of these four accelerators vary significantly, ensuring the AS will provide high performance shared memory for accelerators in all domains.

Each accelerator's RTL is first instrumented to record every memory access, then executes test workloads for each accelerator. This determines the average bandwidth, maximum bandwidth, and bandwidth variation for every memory over full workloads. Bandwidth use for each memory is shown in Figure 3.2.

Each memory's degree of dependency, a measure of latency sensitivity, is also analyzed. Sharing memories adds more logic and latency into each memory access. It is therefore important to



(a) Viterbi decoder

(b) AES-128 encrypter

(c) 1024 point, 16-bit FFT

(d) JPEG encoder, one horizontal line

Figure 3.2: Total memory bandwidth utilization of several accelerators
Plots indicate one round of computation. Labels on the Y-axis indicate maximum binned bandwidth for each memory

ensure shared memories are latency insensitive. A memory is latency sensitive and highly dependent if the results of previous memory accesses contain information necessary to perform the next access. Highly dependent memories with long access latencies exhibit poor performance because logic must stall while waiting for memory accesses to complete. A memory used for linked list traversal exhibits high dependency because the first node must be completely read from memory to determine the memory address of the next node to read. Conversely, memory accesses from non-dependent memories do not depend on previous accesses to complete and can be planned in advance. Therefore, non-dependent memories are insensitive to increased memory latency and amenable to memory sharing. FIFOs are inherently non-dependent because access order is fixed

by design (first in, first out). Many random access memories are non-dependent or can be designed non-dependent by pipelining memory accesses.

Using capacity as well as the above dependency and bandwidth characteristics, each accelerator memory is placed into one of four categories: inter-accelerator FIFO, intra-accelerator FIFO, large internal RAM, and small lookup table.

Inter-accelerator FIFOs are used to load data into the accelerator for processing, or load data out after processing. Accelerators typically communicate with the system over shared buses, which are subject to arbitration delays for every transfer. Larger transmissions are more efficient: reducing the number of transmissions by increasing transmission size results in fewer arbitration delays. These large transmissions are buffered in the inter-accelerator memories, so these memories must be large enough to store the large transmissions. Inter-accelerator memories are typically sized in the kilobytes, and are assumed to be 2 KB for accelerators that do not define inter-accelerator memory size.

Inter-accelerator FIFOs exhibit bursty bandwidths, using their full bandwidth at the beginning or end of an operation when data is streamed in or out. These memories spend many cycles completely dormant after transfers until the next batch of data is ready. This behavior is shown for every accelerator's input and output FIFOs in Figure 3.2.

Inter-accelerator FIFOs do not introduce additional memory dependencies and can be easily pipelined. These FIFOs do not need to follow a strict schedule, provided input FIFOs are filled before the next operation begins and output FIFOs are unloaded before the next operation ends. This timing can be relaxed further if the FIFOs are sized to handle data sets for multiple operations. Inter-accelerator FIFOs make ideal candidates for memory sharing. They consume large amounts of accelerator area, so sharing these memories results in significant area savings. Inter-accelerator FIFOs have relatively light bandwidth requirements. These memories are insensitive to the increases in access latency that memory sharing would introduce because they do not require strict schedules. Sharing inter-accelerator FIFOs also adds the unique advantage of merging input and output FIFOs between accelerators. Intra-accelerator FIFOs feature attributes similar to inter-accelerator FIFOs, though not to the same magnitude. Some accelerators consist of several stages, each resembling a small accelerator. Intra-accelerator FIFOs are used to connect these stages and build up larger accelerators consisting of several steps. For example, the JPEG accelerator uses FI-

FOs to connect Huffman encoding, run-length encoding, and other stages. Accelerators use FIFOs rather than direct connections so stages can be designed independently and to ease timing complexities between stages. Intra-accelerator FIFOs are therefore resilient to memory access latencies and non-dependent. Unlike inter-accelerator FIFOs, intra-accelerator FIFOs are not all large and only some will be worth sharing. As shown in Figure 3.2, bandwidth variations are much smaller in intra-accelerator FIFOs and result in fewer idle periods. Some accelerators require large amounts of bandwidth, and others need only a small trickle (Huffman encoder, RLE). The low bandwidth memories are also the largest, resulting in the greatest area savings and lowest performance impact.

Deciding to share intra-accelerator FIFOs is a more subjective choice than for inter-accelerator FIFOs. Both FIFO types are non-dependent, but intra-accelerator FIFOs may be too small or require too much bandwidth to make sharing worthwhile. Therefore, a bandwidth and capacity based analysis is necessary before considering intra-accelerator FIFOs for sharing.

The third memory type identified is large internal RAMs. These memories are typically sized in the kilobytes and are often used for values that rarely change. For example, the FFT stores constant coefficients for its butterfly operation in four 2 KB SRAMs as seen in Figure 3.2(c). Other memories are used as internal buffers for values that must be written out of order (JPEG JFIF RAM, FFT data RAM). In most cases these memories are non-dependent because access addresses are predictable, and requests to these memories are easy to pipeline. Bandwidth requirements for these memories also tend to be low and bursty. Internal RAMs are large, usually exhibit low or no dependence, and require little bandwidth. Therefore, large internal RAMs are well suited for memory sharing.

The final memory considered is small RAM and ROM lookup tables (LUTs). These are the smallest memories, usually about 100 bytes. These memories typically require high bandwidths, and are often used to determine control flow, making pipelining difficult. Sharing small RAM/ROM LUTs provides little benefit due to their small size and performance overheads.

3.1.3 Shared memory selection methodology

Ultimately, the selection of which memories to share should maximize shared area and minimize performance overheads. Large, low bandwidth, and non-dependent memories are ideal for sharing. At first glance, inter-accelerator FIFOs, large internal RAMs, and some intra-accelerator FIFOs

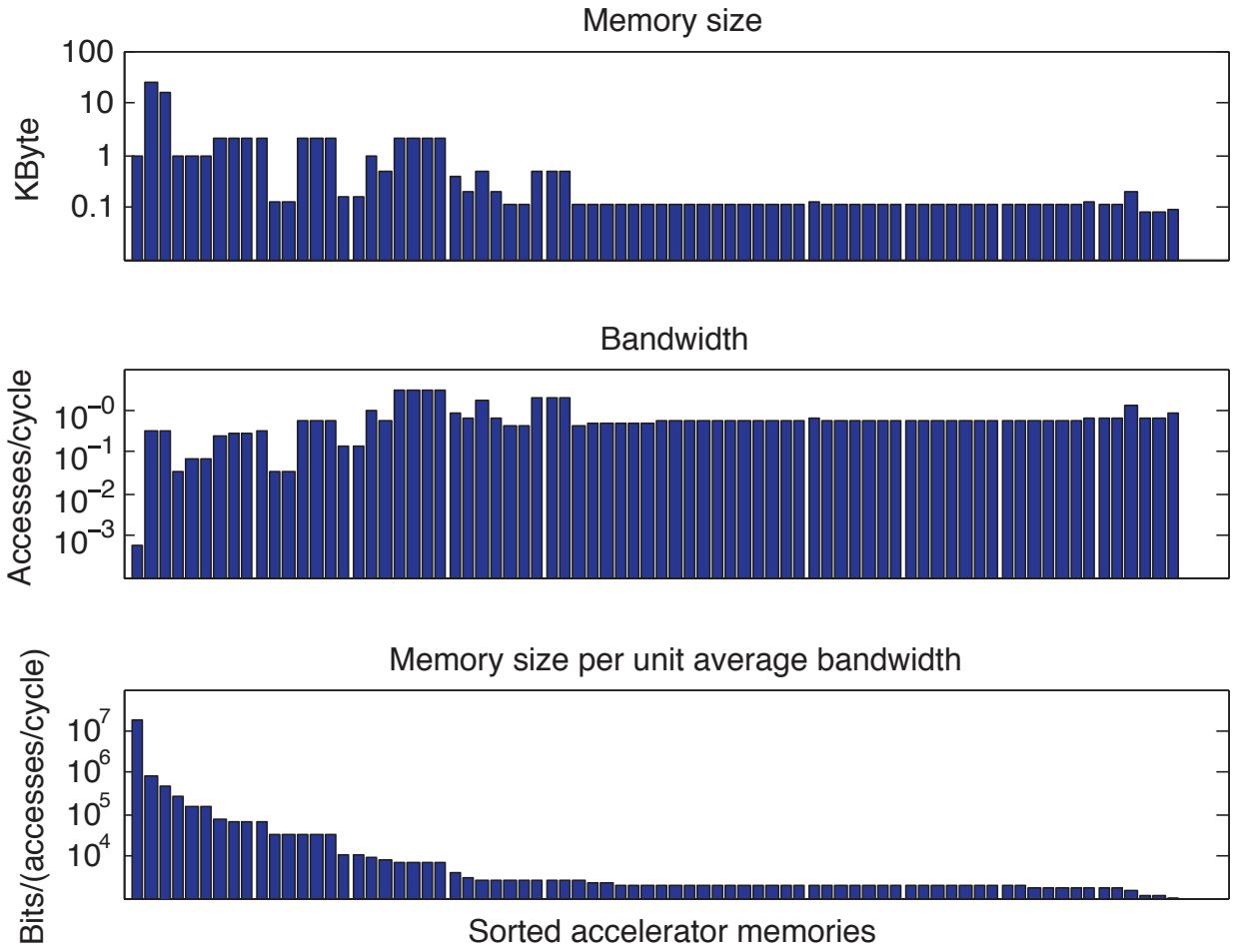


Figure 3.3: Accelerator SRAM memories sorted by memory size per bandwidth

are the best candidates for sharing. To formally decide which accelerator memories to share, the following methodology is suggested:

1. Calculate the average bandwidth of each memory for all accelerators, assuming the accelerator is always processing (no idle periods).

2. Calculate and sort each memory by memory size / bandwidth. This figure provides a balance between maximizing memory sharing and minimizing performance overheads due to contention.

3. Pick memories with the largest memory size / bandwidth and weed out highly dependent memories. These memories strike a balance between area savings and low performance overheads.

This methodology balances area savings and performance overheads effectively. It is also application independent, since each accelerator is assumed to always be active and processing at maximum ability. Optimizing for each accelerator's worst case bandwidth requirements prevents the system from incurring prohibitive performance overheads regardless of the applications running on the processor. This methodology's success for multiple applications is demonstrated in Section 3.3.

When the above methodology is applied to the memories contained by the four characterized accelerators, results reveal inter-accelerator FIFOs, large internal RAMs, and the larger intra-accelerator FIFOs are the most shareable. As Figure 3.3 shows, larger memories also tend to use less bandwidth. The figure is sorted from most sharable (large size, low bandwidth) on the left to least sharable (small size, high bandwidth) on the right. Although the correspondence is not absolute, the memories providing the biggest sharing benefits also tend to result in the lowest bandwidth demands. This factor and the previous analysis showing the largest memories are the least dependent means that near-maximum memory sharing with minor performance overheads is possible.

Memory sharing provides a significant opportunity to slash on-chip area in many-accelerator systems. Using the proposed methodology for selecting shared memories, near-maximum area savings and minor performance overheads is possible. The following section describes the AS architectural framework for sharing memories based on these findings.

3.2 Accelerator store design

This section describes the accelerator store (AS), an architectural framework to support memory sharing in many-accelerator systems. The accelerator store contains SRAMs for accelerators to store internal state or to communicate with other accelerators. It manages how this shared memory is allocated and provides accelerators with access to shared memory. This section describes AS features, the AS's architecture, using multiple ASes for scalability, the accelerator/AS interface, and the software/AS interface.

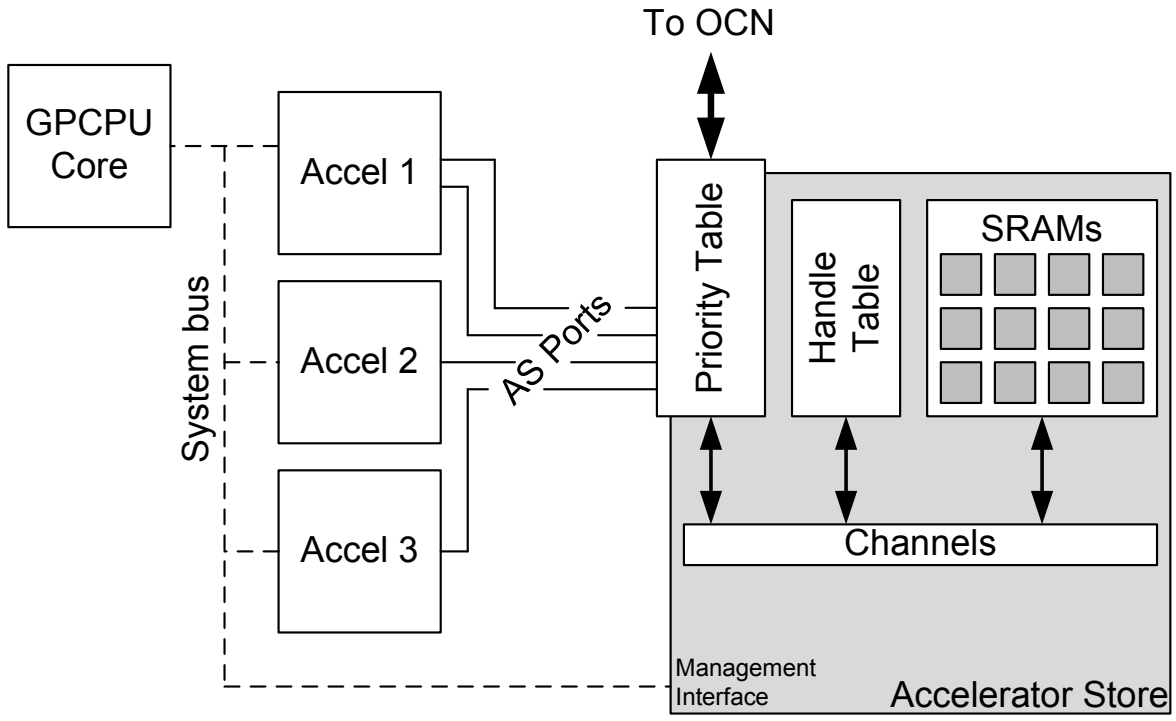


Figure 3.4: Accelerator store system architecture

3.2.1 Accelerator store features

Handles are integral to the accelerator store's ability to share memory with accelerators. A handle represents a shared memory stored in the accelerator store, similar to the way a file handle represents a file in the C programming language or a virtual address space represents a region of memory used by an application. Each accelerator store can contain several shared memories and uses handles to keep track of these memories. To create a shared memory in the accelerator store, the system adds a handle (with a corresponding handle ID) with the shared memory's configuration. The system passes this handle ID (HID) to an accelerator, giving the accelerator access to the shared memory. Accelerators can use multiple handles if more than one shared memory is needed by retaining HIDs for each handle. The system can pass the HID to multiple accelerators, so the accelerators can exchange data through shared memory. Accelerators include the HID when sending access requests to the accelerator store to access shared memory. When shared memory is no longer needed, the system can remove it by deleting its corresponding handle from the accelerator store and informing accelerators that the matching HID is invalid. Specifics about how the accelerator store manages handles are given in Section 3.2.2, and details about how system software can configure handles

are discussed in Section 3.2.5.

Support for handles also allows the accelerator store to emulate multiple types of memories:

- *Random access (RA):* memories allow accelerators to read or write data at any location in the shared memory. RA memories are useful for representing shared internal RAMs.

- *FIFOs:* can only put data in or get data out, in first in/first out order. FIFOs are useful for representing shared inter- or intra-accelerator queues. Support for FIFOs provides a simpler alternative to DMA controllers for exchanging data between accelerators.

- *Hybrid:* memories combine RA and FIFO types, and support reads, writes, puts, and gets. Hybrid addressing maps address 0 to the head value (next value out) and addresses increase moving toward the tail value. Unlike puts or gets, hybrid reads and writes do not add or remove values. Hybrid memories are useful for operations that stream in blocks of data that are used out of order.

The accelerator store keeps leakage power low by VDD-gating unused SRAMs. Initially, all physical memory in the accelerator store is turned off, keeping leakage to a minimum. SRAM memory must be kept on to retain data, so a memory is turned on once it holds any value. Unfortunately, current SRAMs must be completely on or off, and VDD-gating part of an SRAM is problematic. To keep leakage power low, each accelerator store contains several 2KB and 4KB SRAMs rather than one large SRAM. This design allows the accelerator store to save power by turning off SRAMs that do not contain valid data at a finer granularity. If the accelerator store used only one large SRAM, the entire memory would be forced to turn on, even if the SRAM only stored one word.

Once a word is written, shared random access memories must remain on until the memory's handle is removed from the accelerator store. The accelerator store VDD-gates shared FIFO and hybrid memory more aggressively because it can automatically identify memory that does not contain valid data. Once an accelerator gets a value from a shared FIFO, the accelerator store knows that value is no longer stored in the FIFO. Large FIFOs may span multiple SRAMs in the accelerator store, and the accelerator store will turn off any SRAMs allocated to a FIFO if the SRAMs do not contain valid FIFO data.

Table 3.2: Example handle table layout

HID	Alloc (y/n)	Type	Start Addr	Mask (Size)	Head Offset	Tail Offset	Full (y/n)	Trigger
0	Y	RAND	0x0600	0xFF00 (256)	X	X	X	X
1	Y	FIFO	0x0000	0xFC00 (1024)	0x0081	0x0081	Y	224
2	Y	FIFO	0x0400	0xFE00 (512)	0x0010	0x0004	N	0 (off)
3	N	X	X	X	X	X	X	X
...								

To prevent accelerators from accidentally destroying handles mapped to other accelerators, only the general purpose CPU (GP-CPU) can add or remove handles. In most cases, this scheme limits accelerators to modifying handles they are mapped to. Of course, if GP-CPU software incorrectly maps handles to accelerators, or accelerators cannot be trusted to access the handles they are mapped to, accelerators can make unauthorized handle accesses. If more security is necessary, a list of allowed accelerators can be added to each handle in the handle table, blocking accelerators from making unauthorized handle accesses.

3.2.2 Architecture of the accelerator store

The accelerator store depends on several elements as shown in Figure 3.4. A GP-CPU first configures the accelerator store and accelerators over the system bus and AS management interface. Accelerators can then use their ASPorts to send access requests to the AS's priority table. The priority table selects as many requests as there are channels to send them. The channels send these requests to the handle table which translates the request into a physical address for the AS's SRAMs. The channels transmit the physical address requests to the SRAMs and perform the memory access. The channels finally relay the access result back to the accelerator via the channel, priority table, and finally the ASPort.

The accelerator store relies on many small SRAMs to increase VDD-gating opportunities as previously described. Measurements of SRAMs fabricated in a commercial 130nm process revealed that 2KB and 4KB SRAMs provided the best balance between VDD-gating granularity and arbitration overheads.

Handle Table The handle table maintains each shared memory in the AS by storing the handles for each active shared memory (Table 3.2). The handle table stores sixteen handles by default, although this number can be changed if the accelerator store is expected to simultaneously share more than sixteen memories. Each handle includes several pieces of information, including

the shared memory's size, which SRAMs store the shared memory, as well as FIFO-specific settings. After the system configures handles for each shared memory, the handle table's primary function is to translate access requests from accelerators to the AS's SRAMs. For RA shared memory requests that contain a handle ID and offset address, the handle table obtains the SRAM physical address by looking up the handle's starting address and adding it to the request's offset address. This operation is similar to the virtual to physical address mapping performed by MMUs. The translation for FIFO and hybrid shared memories is similar, but uses the head offset for gets and the tail offset for puts.

The handle table also enables the already discussed SRAM VDD-gating. An SRAM can be VDD-gated off if: no handles are mapped to the SRAM, the SRAM contains an untouched RA memory, or maps the SRAM to a FIFO but does not contain valid data. The handle table continually monitors changes to each handle, waiting for one of the above conditions to be true for each SRAM. If so, the handle table signals the relevant SRAMs to turn off and keeps SRAM leakage power to a minimum.

The handle table's trigger allows workloads to be batched, to maximize VDD-gating energy savings and to reduce AS contention. Each FIFO handle maintains a trigger value; when the number of elements in the FIFO reaches this value, the AS raises an interrupt in the GP-CPU. The GP-CPU can then turn on an accelerator to consume the FIFO's data. Setting the trigger value close to the size of a handle's SRAM can reduce leakage power and contention. Rather than turning all accelerators on, which results in under-utilized SRAMs and high competition for AS resources, the trigger allows accelerators to only turn on when a batch of workloads is available, and without turning on additional SRAMs. In addition, fewer accelerators are on at any given time, resulting in fewer simultaneous AS requests, lowering contention and increasing performance.

Channels All shared memory requests from accelerators are transmitted over channels. Each channel can carry one shared memory access per cycle, so ASes servicing accelerators with higher bandwidth needs should provision additional channels. However, additional channels require extra arbitration in the accelerator store and this extra logic will require additional area and power. In addition, additional channels will add more wires and increase dynamic power. If many channels are required for many-accelerator systems, the multiple distributed accelerator stores should be used as described in Section 3.2.3. In the distributed AS model, each accelerator store offers a

few channels to provide a good balance between performance and overheads, which is analyzed in Section 3.3.

Priority Table The priority table controls which shared memory accesses are completed if too many accelerators contend for channels. Each accelerator has at least one ASPort for communicating with the accelerator store, and may have multiple ASPorts for increased bandwidth. Each of these ASPorts is identified by an ASPort ID. The system configures the priority table by assigning a priority to each ASPort ID, so that some ASPorts have priority over others. Assuming the accelerator store has n channels, the priority table will select up to n memory requests from the ASPorts with the highest priorities. This approach can be used to insure that time-sensitive operations take precedence over operations with flexible timing requirements. The priority table can be modified at runtime, so round-robin and other arbitration schemes can be implemented in software.

Management Interface The accelerator store also features a management interface accessible over the system bus, allowing the system to configure the handle table and priority table dynamically at runtime. The management interface is memory mapped, so software can use memory load and store instructions to add or remove shared memories in the handle table, or modify arbitration settings in the priority table.

3.2.3 Distributed accelerator store architecture

As the number of accelerators in many-accelerator systems reaches the tens or hundreds, the accelerator store will grow accordingly. The number of channels and ASPorts required to support hundreds of accelerators will not scale within a single accelerator store; instead, systems should include multiple accelerator stores, as shown in Figure 3.1(c). Each accelerator's ASPort will be directly connected to a single AS and it will primarily use this AS to keep access latencies low. As a result, the system topology will consist of several clusters of accelerators, each cluster surrounding a single AS.

An accelerator may need to communicate with ASes outside of its cluster in some cases. This may happen if the accelerator's primary AS is fully allocated, or if the accelerator must communicate with an accelerator tied to a different primary AS. In these cases, the accelerator will utilize an on-chip network (OCN), allowing accelerator stores to communicate directly. OCNs have been well

studied in the network on-chip (NoC) community [10, 61] and are not duplicated here. Several OCN topologies can be used to connect the distributed accelerator stores, and a grid topology will most likely result in the best scalability.

Although OCNs will introduce additional latency, they will not result in significant performance overheads. Only latency insensitive memories are shared, as discussed in Section 3.1, so the additional OCN latency will not cause stalls or noticeably affect performance. Further, accelerators will use their primary AS most of the time, and will occasionally use other ASes when communicating with accelerators in other clusters. Today's SoCs use high-latency DMA transfers to exchange data between accelerators, so the OCN will not introduce any new high-latency accesses.

Communicating with distributed ASes is simple from the accelerator's viewpoint. To each accelerator, the system contains one AS. This abstraction is achieved by pre-assigning handle IDs to each AS at circuit design time. For example, the first AS would contain HIDs 0 through 15, the second AS would contain HIDs 16 through 31, and so on. To access a shared memory in the first AS, an accelerator would simply use a HID from 0 to 15, regardless of the accelerator's primary AS. Each accelerator store maps HIDs to ASes and will forward access requests over the OCN if the request does not match the primary AS's HIDs. Therefore, the software compiler or dynamic allocation libraries are responsible for assigning AS handles to accelerators in the same cluster whenever possible.

3.2.4 Accelerator/accelerator store interface

Each accelerator communicates with the accelerator store over ASPorts, as previously mentioned. Accelerators may contain one ASPort, or add additional ASPorts to increase bandwidth.

Each ASPort carries three types of messages. First, the accelerator store sends access requests to the accelerator store. An access request contains the type of request (read, write, get, put) and a HID. If the access is a read or a write, the request will also contain an address offset. The accelerator store may not be able to satisfy a request in certain cases and will send the accelerator an access reply. For example, an accelerator may have tried to do a FIFO get on an empty FIFO. If such an error occurs, the AS will send an access reply with an error code describing the problem. Finally, the AS sends an access response back to the accelerator when the access completes. If the access was a read or a get, the access response will contain the accessed data. If the access is a

write or a put, the access response simply indicates the access completed.

The accelerator store is designed to efficiently support memory sharing in a many-accelerator system. The AS is the result of many design choices necessary to keep overheads low and minimize complexity for accelerator designers. However, some changes within accelerators are necessary to use AS memories. Accelerator designers will need to decide which memories are worth sharing, as shown in Section 3.1. Sharing FIFOs should be a simple process, since FIFOs are already designed to be latency insensitive. Sharing large RAMs will require memory access logic to be pipelined, which introduces small design complexities in most cases. Pipelining these memory requests is much simpler than pipelining in CPU data paths, since these accesses cannot cause branch mispredictions or pipeline flushes.

3.2.5 Accelerator store software interface

Software is a critical element in any computer system, and the many-accelerator architecture is no different. The accelerator store fully exposes the handle table and priority table so that more complex shared memory allocation and scheduling schemes can be built up without complex hardware additions. Moreover, a bridge accelerator allows software to modify the contents of shared memory in the accelerator store. The accelerator store's approach to software creates new research opportunities in system software, described below.

The handle table allows an open approach toward shared memory allocation by completely exposing the table to software. Currently, the application software developer manually allocates shared memory to accelerators, creating handles and mapping their HIDs to accelerators whenever needed. In the future, the number of accelerators in many-accelerator systems will scale upward, and manual allocation will become too complex. Instead, software compilers and dynamic memory allocation software libraries could solve this problem.

Compilers can improve memory allocation using an automated static allocation system. Metadata can be written into software so that the compiler knows each accelerator's shared memory requirements. For example, the AES encrypter requires two 2KB FIFOs to operate. Instead of forcing the software designer to allocate these two FIFOs in the AS's handle table, the compiler could do this automatically through macros or programming language primitives.

Ultimately, dynamic memory allocation will provide the most robust approach to allocate

shared memories. Similar to malloc(), software libraries can keep track of the contents of each accelerator store and allocate shared memory on demand at runtime. This approach uses shared memory more efficiently, because the system will have a better understanding of what memory is available at runtime.

The priority table's full software accessibility can enable more complex scheduling schemes as well. By default, the priority table arbitrates by always picking certain ASPorts over others. To implement a round-robin scheme, the priority table is rotated periodically, moving the lowest priority ASPort to the highest priority slot and moving the other ASPorts down one slot. This gives each accelerator equal time as the top priority slot and eliminates any contention-related starvation. A real-time scheduler could also be implemented by coordinating the priority table with software. If a certain accelerator needs guaranteed access to the accelerator store for a fixed period of time, software could place that accelerator's ASPort at the highest priority for that period of time. Although the priority table's initial configuration is simple, software can add more complex and robust schedulers.

The many-accelerator system contains one or more GP-CPUs, which may wish to access shared memories in the AS. To enable this, a bridge accelerator can be used to bridge the GP-CPU's system bus with the AS. The bridge accelerator is memory mapped on the system bus, enabling the GP-CPU to access shared memory in the accelerator store by writing commands over the bus. The bridge accelerator will replay these access requests to the AS. From the AS's viewpoint, the bridge accelerator is no different from any other accelerator and can be prioritized in the priority table. When the access completes, the accelerator store will return the access response, and the bridge will relay the result back to the GP-CPU.

The bridge scheme separates AS memory from the GP-CPU's system memory. Future systems could integrate these SRAM memories in the same address space, but doing so is beyond the scope of this paper. Such a change would require managing simultaneous accesses to the same SRAM from both accelerators and the GP-CPU, as well as complications to SRAM VDD-gating and caching.

The accelerator store is carefully designed to make sharing memory as simple as possible and keep performance and energy overheads minimal. In the following section the accelerator store's ability to satisfy these goals is evaluated.

3.3 Accelerator Store Evaluation

This section evaluates the accelerator store's ability to reduce area while keeping performance and power overheads low for two applications. The first application is representative of embedded systems and utilizes several accelerators processing a highly serial dataflow. An alternative application representing a server workload utilizes several JPEG encoders operating in parallel. The implementation of the accelerator-based model is presented first before evaluating the benefits and overheads of sharing accelerator memories.

3.3.1 Accelerator-based system model

Six accelerators in the application are modelled: AES, JPEG, FFT, a digital camera, an ADC, and a flash memory interface. The first three accelerators are derived from real accelerators obtained from OpenCores and Xilinx in Table 3.1. Each SRAM memory in the accelerator is instrumented to record every read, write, put, or get at every cycle. The resulting traces allow are used to simulate the accelerator access patterns with contention and latency by playing back the memory access traces in order, and maintaining the accelerator's timing between accesses. The analysis optimistically assumes that the accelerator store can satisfy accesses in a single cycle.

After optimizing the accelerator store configuration to minimize contention, the evaluation demonstrates the system's access latency tolerance by increasing the accelerator store's access latency up to 50 cycles. Although the analysis assumes that each memory inside the accelerator is non-dependent as described in Section 3.1, accesses between accelerators are still treated as dependent. As an example, consider the case where accelerator A wishes to send data to accelerator B. Accelerator A must first completely write its data to shared memory before accelerator B can attempt to read it. To ensure this dependency, accelerator B stalls until the write completes.

RTL implementations for the remaining three accelerators were not available, so memory access traces were manually generated via known device characteristics. The camera is modeled after a synthetic camera found in the JPEG testbench, the ADC will write one 16-bit word at 44.1 KHz, and the flash accelerator writes data in 2 KB pages at 6 Mbps, typical settings for an SD flash card [64].

The accelerator store is implemented in Verilog RTL, and the number of channels is parame-

terized. The design was synthesized with Design Compiler and placed-and-routed in Encounter for a commercial 130nm process. The AS requires an area overhead of 80,000 μm^2, equivalent to 1% of the area used by the six accelerators from the embedded application.

The simulation model consists of three stages. Scheduling is done first to decide when each accelerator will begin and complete an operation. This step is done without considering contention, as if the accelerators do not share memory. Memory access overlay is performed second, to copy the memory access traces obtained from real accelerators into the schedule. This step is also completed as if no memory sharing is possible. Stalling replay, the final step, considers contention due to shared memory. The access logs are replayed for each accelerator and accelerators may stall if more requests are made than accelerator store channels are available. Each accelerator tracks its own time in addition to the system time; accelerators may only increment their cycle count if all pending accelerator store requests have been satisfied. Accelerators are also able to fast forward past cycles in which the accelerator would be turned off. This feature models the situation where an accelerator stalled for a few cycles, but finished its operation and can catch up to the rest of the system.

Unfortunately, it is not possible to evaluate systems with multiple accelerator stores due to the difficulty of obtaining a large number of distinct accelerators. Instead, the evaluation demonstrates one accelerator store/accelerator cluster, and artificially increases memory access latency to simulate the additional latency the on-chip network would add.

3.3.2 Embedded application

The first application is designed to demonstrate a typical workload for a mobile, embedded device. This application combines the six accelerators previously described to implement a security monitoring system. The ADC samples a microphone at 44.1 KHz, enough to accurately capture frequencies audible to the human ear. These audio samples are processed by the FFT (1024 point, 16-bit, radix-4) every 1024 samples. The resulting frequency response is checked by the GP-CPU to look for a specific frequency, such as a car horn or glass breaking. If such an event is detected, a camera will take a picture once per second. The resulting image is compressed via the JPEG accelerator, then encrypted by the AES accelerator. The encrypted JPEG photos are finally stored in the flash accelerator.

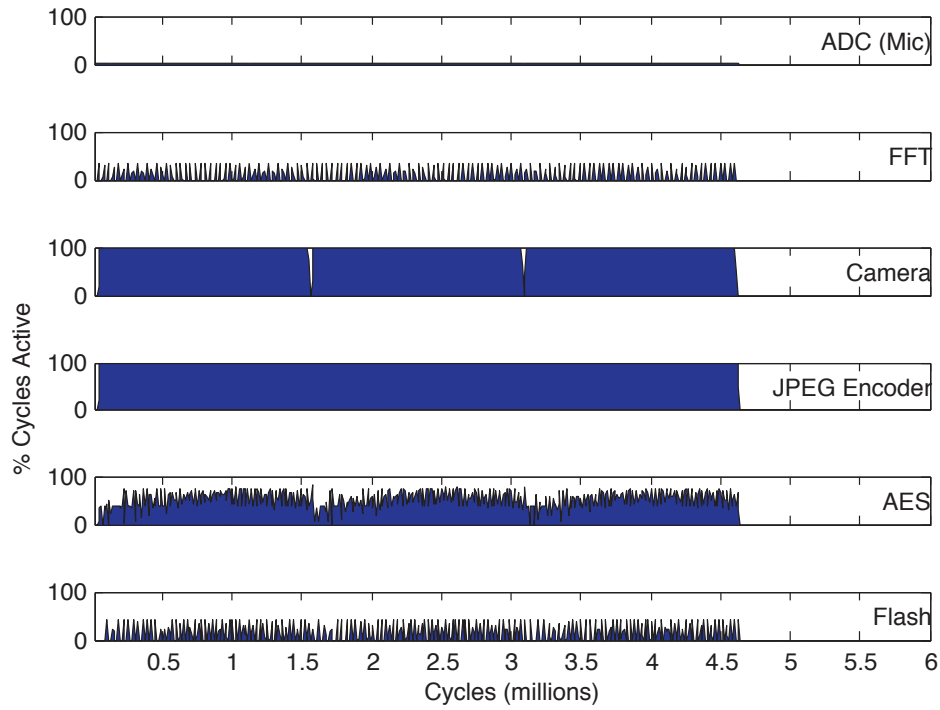


Figure 3.5: Embedded application accelerator activity

Embedded application performance

To model the most contentious workload possible, the frequency is reduced so all accelerators can perform their tasks while meeting the timing guarantees described above. The JPEG accelerator is the limiting accelerator, so the clock frequency is set just fast enough to compress one 640x480 photograph per second (1.53 MHz). Although this clock frequency may seem slow, accelerators can compute far more per cycle than a corresponding GP-CPU. Each accelerator is checked for activity at every cycle and put into one of 100 bins. Figure 3.5 demonstrates how often each accelerator is actively performing work during each of the cycle bins. The JPEG, AES, and camera accelerators are active all or most of the time, and the FFT and flash memory are moderately active. The ADC samples rarely by comparison.

Contention can be kept low despite utilizing six accelerators simultaneously (many at a high duty cycle). Figure 3.6 shows the twenty most shareable accelerator memories (large memory, low bandwidth, non-dependent) as sorted in Figure 3.3. The most shareable memories are drawn bottom to top, and results are grouped into 100 bins and averaged. This figure demonstrates that most of the memories selected by the memory size/bandwidth metric require low bandwidth and

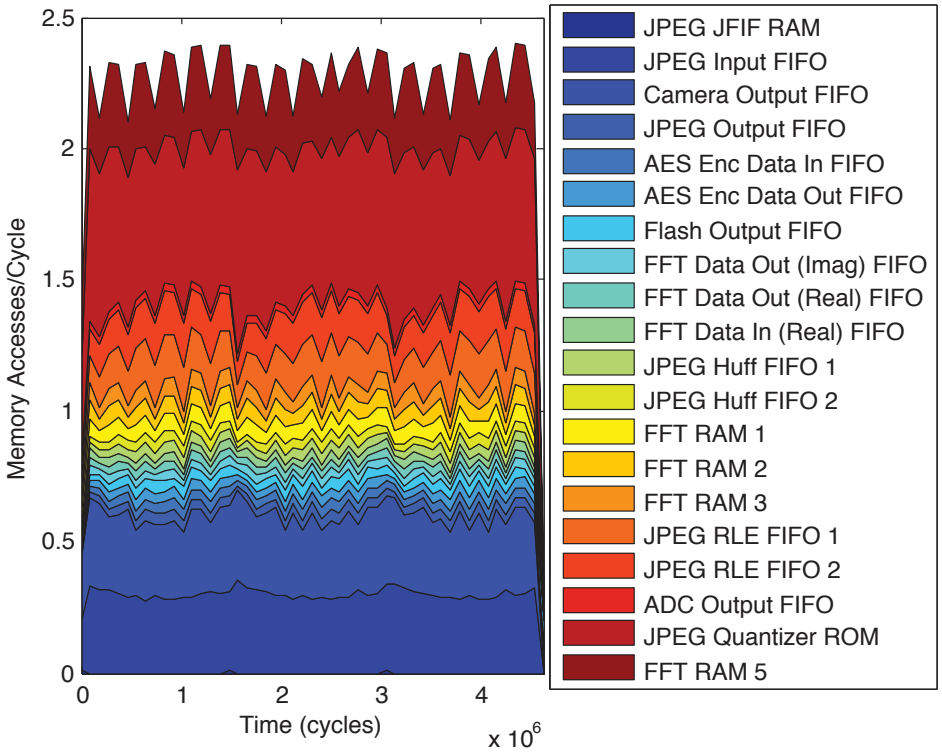


Figure 3.6: Embedded application "top 20 to share" memory bandwidth

the remaining few are quite large (JPEG input FIFO, camera output FIFO). Including at least the top 15 memories would result in low bandwidth contention (only two channels are necessary) and a large percentage of accelerator memory sharing (76%). Up to roughly 25 memories can be shared before performance becomes unacceptable for any number of channels, validating the memory selection methodology from Section 3.1.3.

Although it would not be feasible to fabricate a chip specifically for this embedded application due to design costs, it is important to gauge how effective the sorting and memory selection algorithm would be if performed for the application rather than the application-blind approach. Even if memories are sorted based on their application bandwidth needs and chose these memories based on application contention plots, the result is a negligible reduction in contention and only 2% improvement in memory savings.

The number of accelerator store channels (number of simultaneous accesses) to provision is highly dependent on the processor's design goals. As Figure 3.7 shows, additional channels can significantly alleviate performance overheads. Most applications will add additional channels because the area overhead of additional channels is low. Each additional channel requires approximately

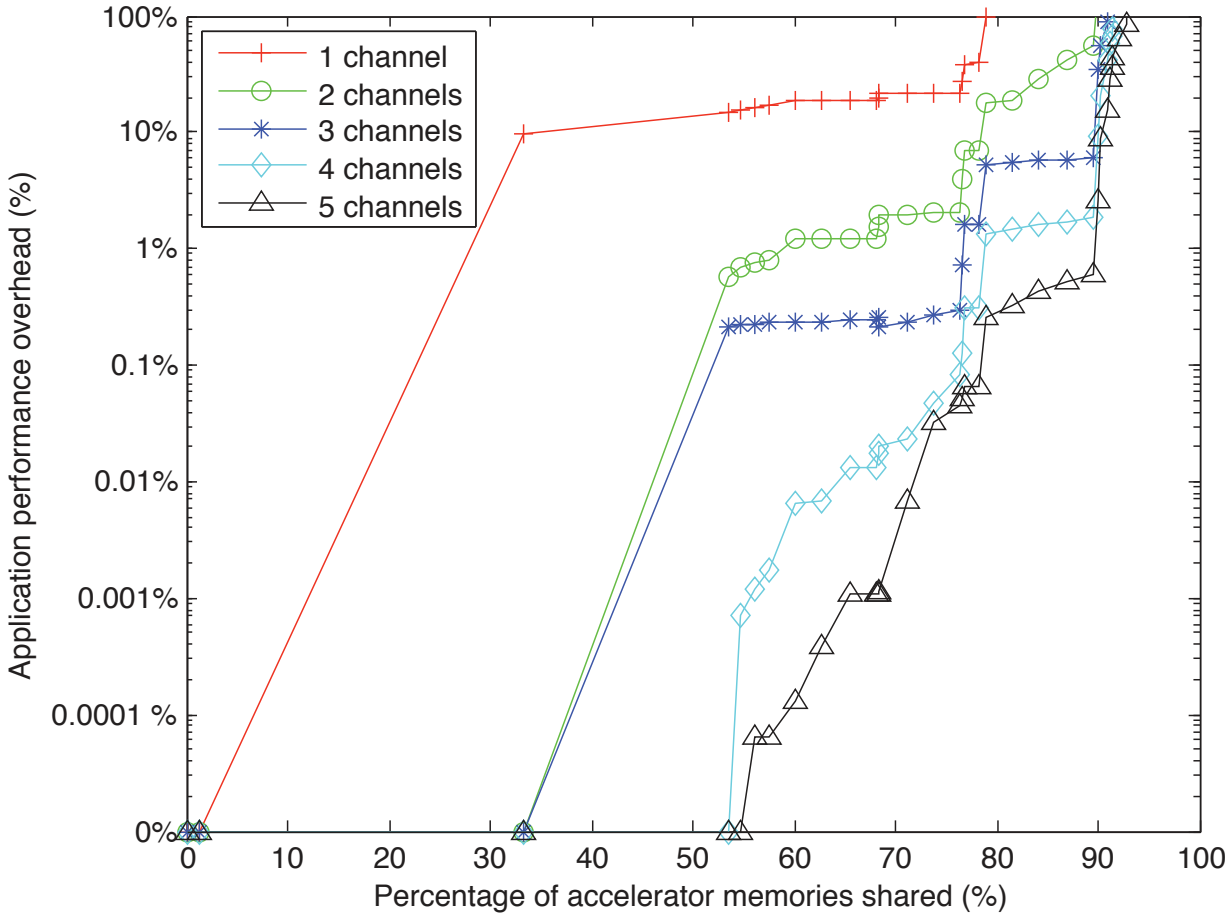


Figure 3.7: Embedded application contention performance overhead

1% of the total system area. If fewer channels are used, the processor will need to operate at an increased frequency (and voltage), resulting in a cubic increase in power. For most systems, this power increase would be unacceptable, though some embedded sensing applications with low duty cycles may not have these performance concerns. For this reason, it is necessary to provision at least three channels for all but the simplest processors.

Distributed Embedded Application Performance

To test the system's latency sensitivity introduced by the distributed AS OCN, the latency of each AS access is increased up through 50 cycles. Because far fewer distinct accelerators were available than one would expect to find in a many-accelerator system, a 50 cycle latency is used to reflect unusually high delays to model such a many-accelerator system. The 50 cycle latency incorporates

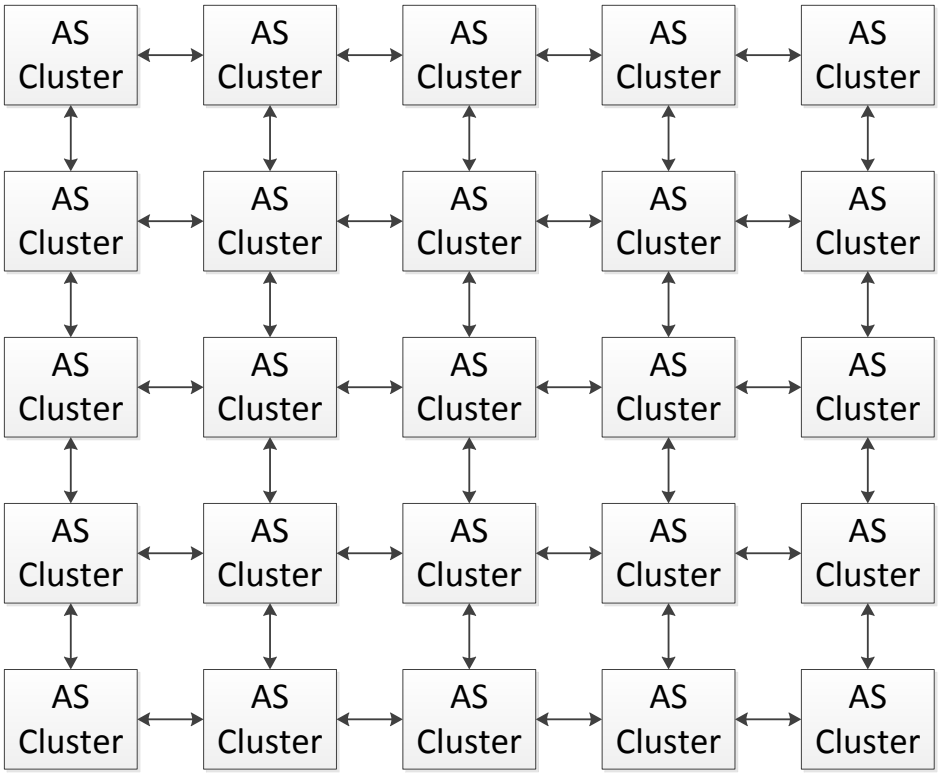


Figure 3.8: Distributed AS architecture
Each AS cluster in the example system contains an AS and several accelerators.

delays due to inter-cluster network routing and contention, delays turning on VDD-gated SRAMs, and delays from AS arbitration.

To demonstrate why 50 cycles represents a unusually high delays in a many accelerator system, a system consisting of 25 AS clusters, shown in Figure 3.8, is considered. The clusters are connected in a 5x5 network-on-chip (NoC) grid. Assuming each cluster resembles the previously modelled single cluster system for the embedded application, each will contain roughly six active accelerators and several others turned off. Therefore, this example system contains 150 active accelerators and many more turned off.

A 50 cycle AS access latency is an unusually high delay in this example system. Assuming

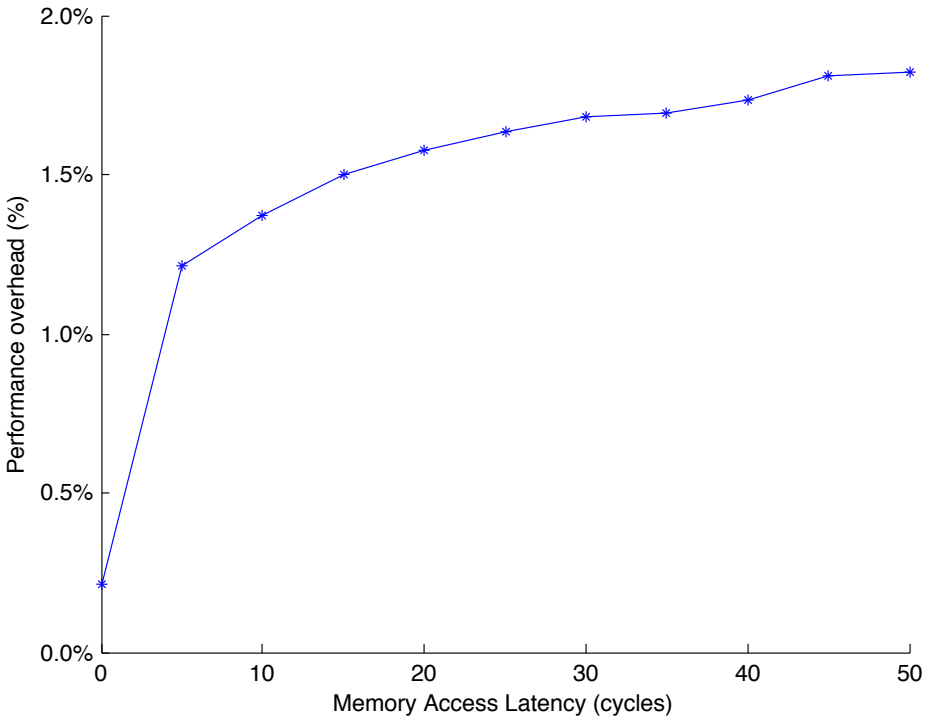


Figure 3.9: Embedded application access latency and contention performance overhead

each hop between clusters requires a cycle, a delay found in commercially available processors [28], the worst case round trip path requires 16 cycles. Using HSIM simulations of 2KB and 4KB SRAM HSPICE models, worst-case SRAM power-on times of 20.5 ns was measured, or 21 cycles when operating at a 1 GHz clock frequency. Of the 50 cycle latency, the remaining 13 cycles are budgeted for AS arbitration (up to 2 cycles) and contention within the NoC. Note that most AS accesses are expected to require fewer cycles since most accesses will not travel the maximum path or require SRAMs to power on. Rather, the 50 cycle latency demonstrates the distributed accelerator store design performs well, adding up to a 2% performance overhead under unusual cases.

These low performance overheads are possible because of careful selection of shared memories, described in Section 3.1.3. If *all* accelerator SRAM memories were shared in the AS, performance overhead would likely exceed 100% due to data dependency delays. Instead, memories are selected with preset access patterns to share in the AS, such as I/O buffers and FIFOs. These memories are the majority of accelerator SRAM area, and their access patterns are known in advance and can be pipelined. As a result, the performance overhead for accessing these memories remains below 2%, even for many-accelerator systems.

Embedded Application Power

The accelerator store takes several steps to achieve a low power overhead. To measure accelerator power as well as accelerator store overheads and benefits, several factors are modeled:

- *Accelerator and AS power (active and leakage):* leakage power is estimated using area to power ratios derived from previously synthesized logic at a commercial 130nm process.

- *Wire activity:* All wires used to connect accelerators to the accelerator store are conservatively assumed to be the system's length (2.354 mm). Using known characteristics [37], wire power is estimated as 0.49 pJ/bit/mm at 1.2 V operating voltage.

- *Automatic AS SRAM VDD-gating:* VDD-gated memories are assumed to consume negligible power.

- *Accelerator VDD-gating:* Sharing memory via the AS can make accelerators easier to VDD-gate as well.

The accelerator store was configured with three ASPorts and 15 shared memories as suggested in Section 3.1. The embedded application is executed at a maximum workload for roughly three million cycles.

The accelerator store is designed to keep power costs low. As shown in Figure 3.10, the overheads of the accelerator store add an additional 8% to the total system power cost. The majority of this overhead (5.24%) is incurred by the accelerator store arbitration logic. Additional wire power (2.38%) and accelerator stalling (0.14%) use measurably less power.

The accelerator store is also able to reduce power consumption through aggressive VDD-gating. Although each accelerator could implement VDD-gating individually, accelerators that VDD-gate their memory or logic are rare in practice.

The accelerator store makes VDD-gating SRAMs guaranteed and automatic. Shared memory automatic VDD-gating reduces power by 8.44% by VDD-gating the 24.78KB of SRAMs that are temporarily unused on average. This may make VDD-gating accelerator logic easier as well, since 76% of accelerator memory is shared in the accelerator store and no longer private. An additional 8.81% of power can be trimmed by gating accelerator logic and leaving the remaining private

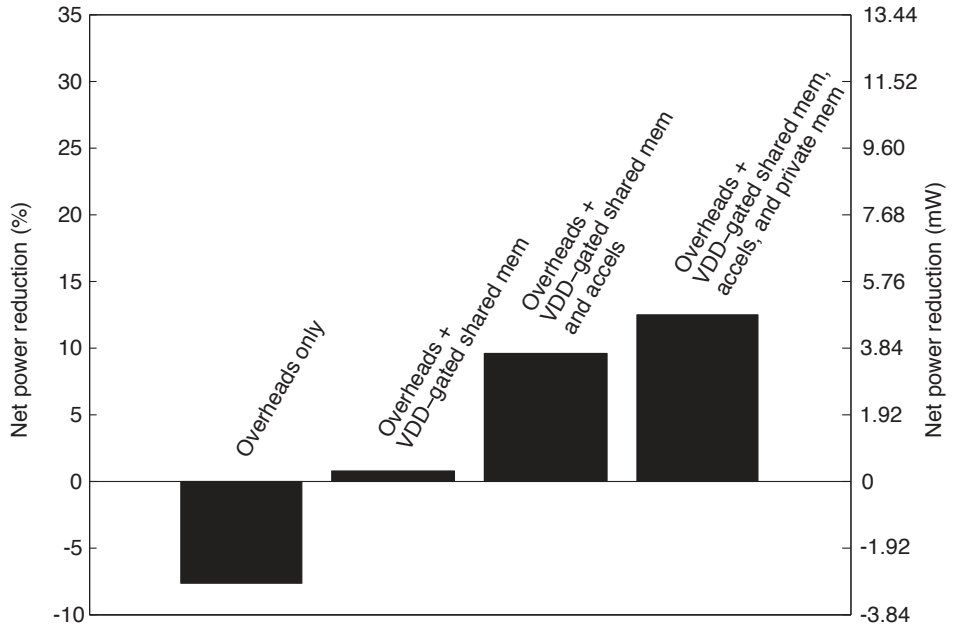


Figure 3.10: Embedded app power breakdown

memories on, or 11.76% of system power can be saved by VDD-gating accelerator logic and private memories.

Leakage power is a growing concern as fabrication technologies continue to shrink. ARM has noted that low leakage technologies that SoC designers relied on will no longer be the magic bullet when entering 45nm and smaller technologies, stating, "Whatever transistor is used, leakage management is a significant challenge that must be addressed," and noting 20% increases in performance will result in 1000% increases in leakage [58]. Low-leakage processes also require more active power, and judicious use of VDD-gating may be a better alternative to low-leakage flavors by keeping leakage and active power low. Therefore, the accelerator store's support for automatic VDD-gating will become more critical in the future.

Embedded application area

By following the guidelines proposed in this paper, the processor can share 76% of its accelerator memory and keep power low, all with a minor 2% impact on performance. Until now, only accelerators in use have been considered. As Figure 3.11 demonstrates, memory sharing translates into significant memory savings. The figure starts with six accelerators and a three channel accelerator store, corresponding to the case where all accelerators and all memories are in use, including private

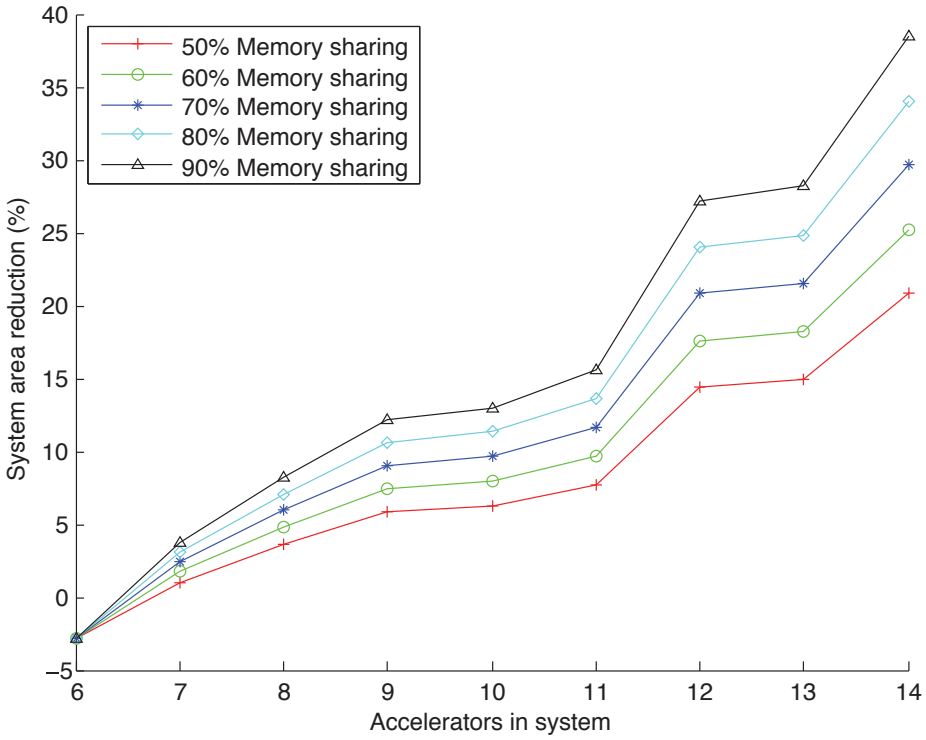


Figure 3.11: Embedded app area reduction

and shared memories. In this case, there is no memory savings from unused memory amortization, and a small area overhead of 3% resulting from the accelerator store area overheads. If a system with more dark accelerators is considered, area savings grow large quickly. The analysis continues by adding unused accelerators to the system as arranged in Table 3.1. If assuming 70% of accelerator memory can be shared on average, total system area is reduced by 30% (this area includes SRAM memory, accelerator logic, and accelerator store logic overhead). Area savings will grow further if larger regions of dark accelerators are included.

Embedded application software implementation

The embedded application is easy to implement in software using the accelerator store. The first step is to allocate memory for the accelerators as needed. This is done by modifying the handle table through memory mapped I/O. For example, a 4 KB FIFO should be allocated for the ADC to put audio samples and for the FFT to get samples for frequency analysis. Although each frequency analysis requires 2 KB of data, creating a 4 KB FIFO allows the ADC to record samples even when the FFT has not finished computation. Note that this single allocation is much easier than

existing buffer designs which would require statically sized buffers in the ADC and on the FFT, and routine DMA operations to transfer data between the two. Using the accelerator store, the system could later do a 512 point FFT and reduce memory buffer size to 2 KB. This software-based memory resizing would not be possible under current designs, since private buffers must be statically allocated at circuit design time.

Second, the priority table must be configured through memory mapped I/O. Memory for recording audio samples and photographs receives a high priority to ensure sampling is performed at strict time intervals. Compression and encryption are less time sensitive, so memories used for these tasks receive lower priorities.

Third, software needs to initialize interrupts and implement interrupt handles for events occurring during accelerator operation. For example, the handle table should be configured to trigger an interrupt when the number of samples stored in the ADC FIFO reaches 1024, enough to perform an FFT. A short interrupt handler would turn the FFT on, pass the ADC FIFO's handle to the FFT (so it knows where to read samples from), and activate the frequency analysis operation.

Software design for accelerator-based tasks in the many-accelerator framework is simple. Most of the computation is performed by accelerators, so software takes on an accelerator management role.

3.3.3 Server application

The sample application representing a server workload is highly parallelized, unlike the serial workflow found in the embedded application. This server application features multiple JPEG encoders compressing separate images in parallel. This application could be used by Facebook, Flickr, or other photo web sites, which must compress many pictures in parallel.

The level of parallelism is swept from one JPEG encoder (no parallelism) to four JPEG encoders (4x parallelism) to examine the accelerator store's performance in a server-class application. Each JPEG's workload is staggered, started slightly after each other to prevent simultaneous bandwidth surges. As Figure 3.2(d) shows, the JPEG accelerator's bandwidth requirements are highly variable.

The server application will require more channels than the embedded application due to the increased use of the JPEG encoder. The JPEG encoder is quite demanding and requires the most

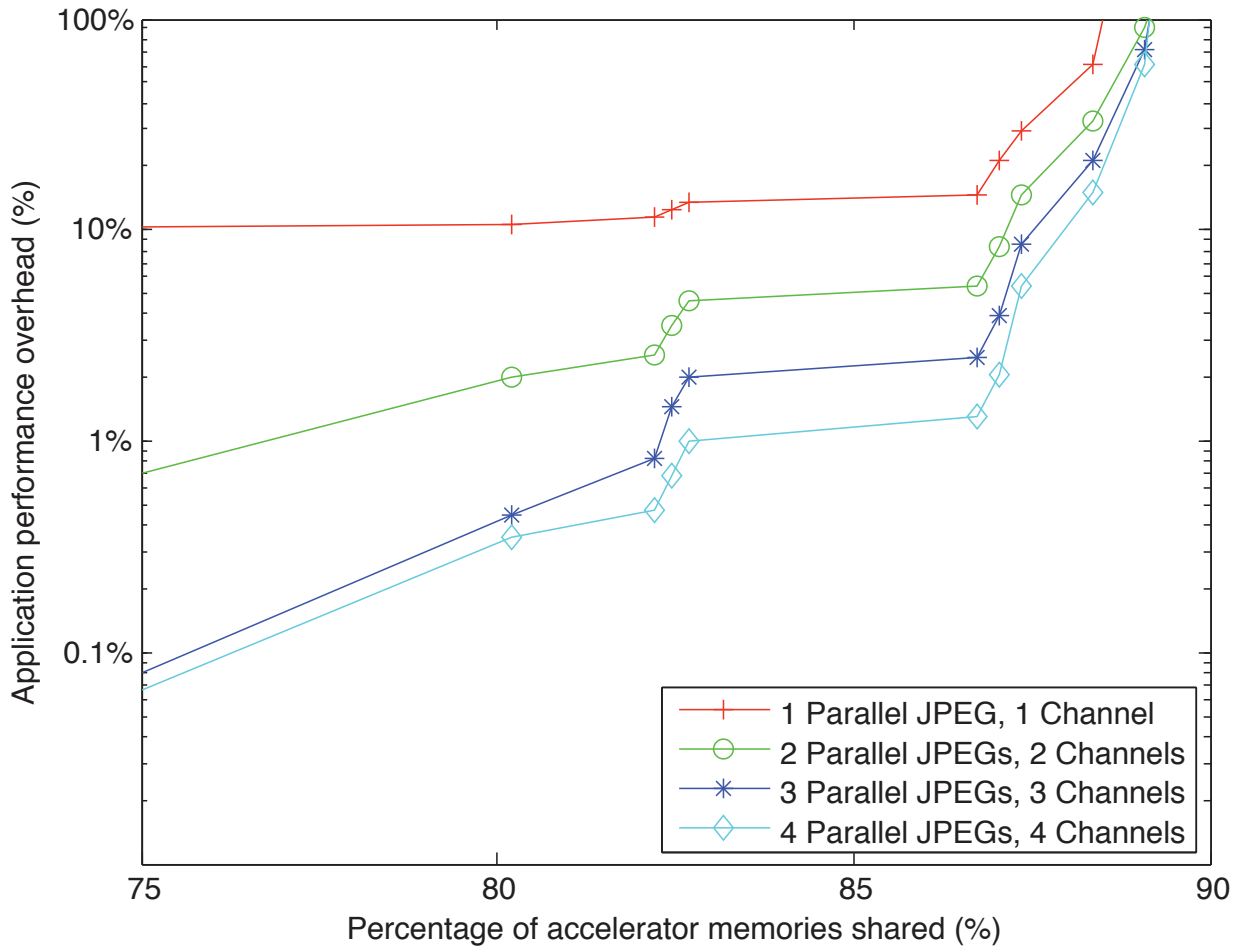
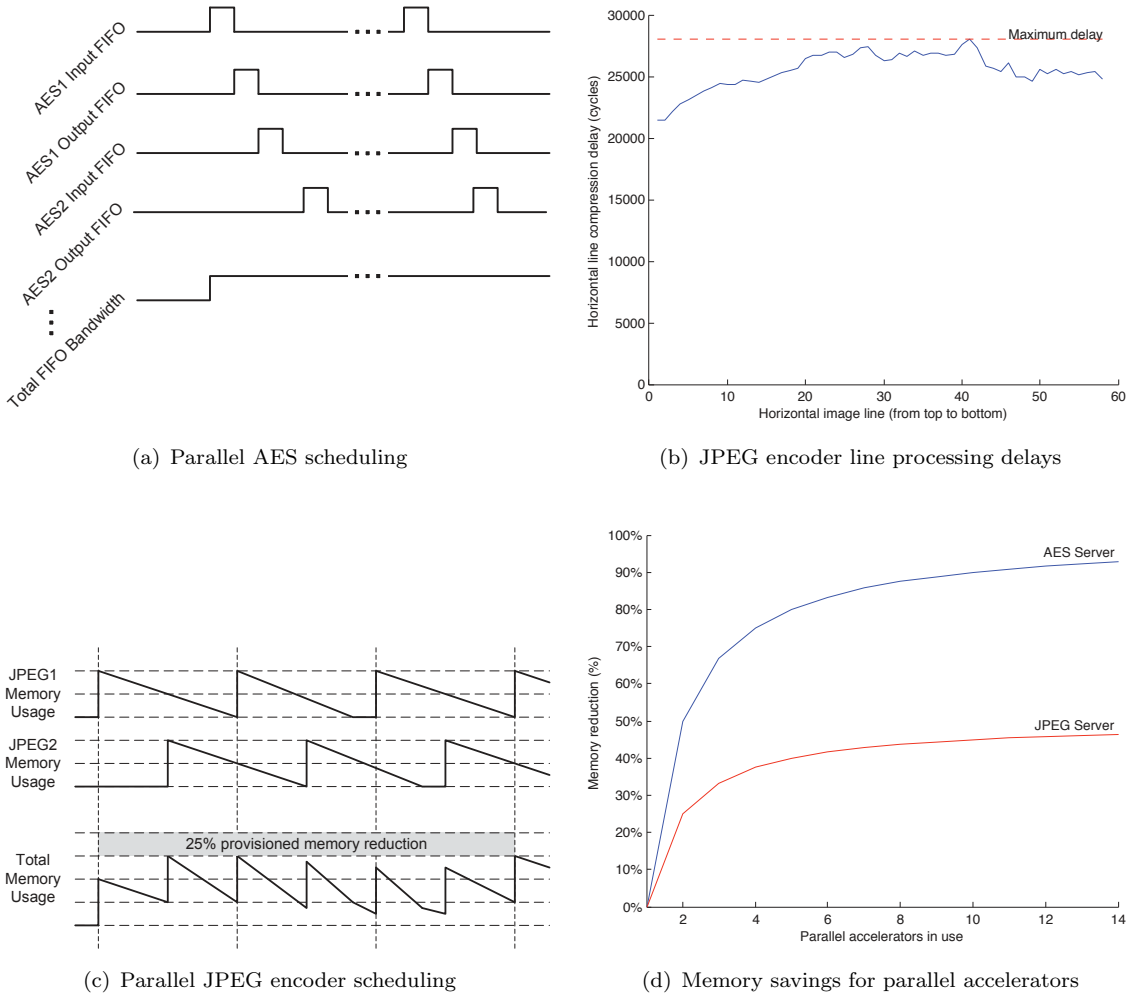


Figure 3.12: JPEG server application performance overhead

bandwidth of the investigated accelerators. Therefore, one channel is provisioned for every JPEG encoder operating in parallel. Processors designed for server workloads are expected to provision more channels than processors for embedded applications.

Including more copies of the accelerator in parallel results in better performance, as shown in Figure 3.12. If considering a JPEG encoder with one ASPort, the performance overhead would be 10% for any significant amount of memory sharing due to the JPEG's bandwidth demands. By staggering each JPEG's execution, each JPEG accelerator rarely demands maximum bandwidth at the same time and the increased bandwidth is amortized between the JPEGs. As a result, the 4x JPEG configuration is able to share more than 85% of its memory with less than 1% performance overhead. This result shows that it is best to include many accelerators under the accelerator store to amortize channels and improve performance.



(a) Parallel AES scheduling

(b) JPEG encoder line processing delays

(c) Parallel JPEG encoder scheduling

(d) Memory savings for parallel accelerators

Figure 3.13: Comparison of architecture styles

Parallel workload scheduling optimizations

Well-scheduled workloads can greatly improve performance and reduce memory provisioning in parallelized shared memory systems. In the server application, the channels are amortized by staggering JPEG execution to smooth performance requirements for the multiple accelerators. This technique can be taken further to support multiple accelerators with less provisioned memory and little or no performance impact.

The gains of a staggered AES server are considered first. The AES accelerator consists of four RAMs (two data, two constants) and two I/O FIFOs. The previous memory selection analysis showed that the two I/O FIFOs are shared but the remaining four RAMs should remain private due to their high bandwidth requirements. Each of these FIFOs is only in use for 8 out of the 117

cycles (7%) required to encrypt a single 128-bit block. The bandwidth needs required by a server utilizing multiple AES accelerators in parallel would increase linearly if each accelerator is started at the same time. By staggering the accelerator workloads and ensuring only one FIFO is active at a time, seven AES accelerators can run in parallel using one channel without any contention or performance overheads. Figure 3.13(a) demonstrates how the schedule is formed to implement this approach.

The well-scheduled AES server can also reduce the required amount of provisioned shared memory. Accelerators relying purely on private memories must include two I/O FIFOs for each AES accelerator. Accelerators relying purely on private memories must include two I/O FIFOs for each AES accelerator. Shared accelerator systems can relax this restriction and share the same I/O FIFOs. The staggered schedule approach ensures one AES accelerator will read in a 128-bit cycle block, the next block will be read by the next accelerator, and so on. As a result, there is no need to provision I/O FIFOs for each accelerator and the 7x AES server can reduce its memory provisioning requirements by a factor of seven as well. If the system has access to a second channel, up to fourteen AES accelerators can stagger their inputs while guaranteeing accelerators will not try to access the I/O FIFOs at the same time. Therefore, this staggered approach can reduce memory provisioning requirements by 14x (26 KB), significantly reducing system area.

The evaluation now considers the more complex staggered JPEG server. As previously observed, staggering can improve performance by 10x for the JPEG server. A 2x memory area reduction of the extremely large JPEG input FIFO is also possible in return for a small performance overhead of 9%. The JPEG server is more difficult to schedule because the delay to perform a single round of processing (an 8-pixel horizontal line of an image) is not constant, unlike the AES accelerator. Figure 3.13(b) shows the distribution of processing delays over the compression of an image. To maintain a consistent schedule the time to compute each line must be constant, so each line requires the maximum time indicated by the distribution of processing times. As a result, lines which require less than the maximum time must be stalled, causing the 9% performance overhead.

The JPEG input FIFO's memory usage profile must be understood to minimize memory provisioning requirements. The JPEG algorithm processes the images in 8x8 pixel squares. This presents a problem as uncompressed images are typically rasterized, drawing a horizontal line one pixel thick at a time. The accelerator store can alleviate this problem through support for hybrid

memories (random access and FIFO). The accelerator providing the uncompressed image (camera, for example) can use random access to rearrange the image so the pixels are arranged in order of the 8x8 pixel squares required by the JPEG accelerator. The JPEG accelerator can use the same handle as a FIFO to read out the pixels in processing order. Now that a FIFO is used, the accelerator store can turn off memories as the contents of the FIFO shrinks. This reorganization technique will not result in any memory provisioning savings since the maximum required memory for the input FIFO will remain enough to store at the 8-pixel wide line. However, the maximum memory requirements can be reduced when several JPEG accelerators use a staggered parallel schedule.

As a result of the FIFO reorganizations made possible by the accelerator store, the input FIFO will immediately use the space required by the 8-pixel tall line and linearly reduce the memory consumption until the line is processed. Through effective staggering, the sawtooth pattern in Figure 3.13(c) can be exploited to reduce the maximum memory utilization of the JPEG encoders when considered in sum. For example, when two JPEGs are operating in parallel, the second JPEG accelerator will be scheduled to begin processing a line when the first accelerator has compressed 50% of its line. Under this case, the first accelerator's input FIFO will require enough memory to store half a line while the second JPEG encoder needs enough memory to store a full line, resulting in a 25% reduction in memory provisioning requirements. If a three JPEG encoder is used, the staggering schedule will result in the input FIFOs requiring memory to store 33%, 67%, and 100% of a 8-pixel wide line, corresponding to a 33% reduction in area. As Figure 3.13(d) demonstrates, memory provisioning reductions approach 50% (168 KB) as more JPEG encoders are used, resulting in significant system area reductions.

3.4 Related work

As mobile computing has become pervasive, SoCs have been developed to provide general purpose and application specific processing on one platform. In one example system, separate scratch pad memories are included within a security accelerator that must be managed by the application programmer or software library [8]. Brick and mortar assembly techniques could provide a low cost approach to piece together accelerator-based architectures without requiring a new lithographic mask for each collection of accelerators [43]. This work does not address the need for dynamic

allocation of memory for each accelerator, nor does it specify an interface to communicate between accelerators without copying data between accelerators. Programming environments for heterogeneous systems have been proposed which could be applied to the accelerator store [48]. These library based frameworks are used to map general purpose code down to graphics accelerators.

Supporting advanced memory management in hardware for accelerator based architectures has been explored by a few groups, but without the full features provided by the accelerator store. An SoC dynamic memory management unit (SoCDMMU) has been proposed for multicore SoCs that provides support for malloc() in hardware [65]. This system does not support advanced structures (FIFOs, hybrids), automatic VDD-gating, or handles. Smart memories features a general purpose microcontroller in multiple tiles that interact with memory [52]. Unlike smart memories, the accelerator store targets heterogeneous accelerated systems and includes support for automatic memory VDD-gating.

Sharing memory within functional units in general purpose processors has been investigated [55]. This work optimizes shared memory between GP-CPU blocks (L1, BTB, etc.) using two tiered caching. Disaggregated Memory shares memory between server blades [46]. In contrast, the AS introduces novel features designed for accelerators including FIFO support, improved inter-accelerator communication by eliminating redundant copies, automatic VDD-gating, configurable priorities, and highly configurable memory management.

Chapter 4

ShrinkFit

Reconfigurable FPGA logic has grown into a multi-billion dollar industry by enabling accelerator-based system designs without the high initial costs and long production times of ASICs. Despite the potential for performance boosts, the difficulty of creating accelerators in Verilog or VHDL RTL has been a lingering challenge. Advances in high level language (HLL) synthesis tools show new promise in simplifying this design process. More accessible languages, such as C, can now be used to create and test hardware accelerators in less time. By reducing barriers to entry, these HLL tools can bring hardware accelerator design to a wider audience of developers.

Commercial hybrid processors, combining FPGAs and general purpose cores [5, 2], have the potential to broaden the audience for acceleration as well. Using hybrid systems, programmers can package accelerator designs with applications, and improve application performance by deploying accelerators to FPGA fabric. This enables systems to run multiple applications, each deploying and using one or more accelerators.

One such system is a prototype of the electronic "brain" of a flying robotic bee, called RoboBee. This hybrid processor prototype combines a general purpose Cortex-M0 core with accelerators on a Spartan-6 FPGA, and runs a bee application. During the development of this prototype processor, HLL tools were invaluable in easing the process of implementing accelerators. However, as much of the FPGA community has found [38, 25, 41], combining accelerators into a single system is difficult.

Current approaches for designing systems containing multiple accelerators often use HLL tools to create multiple variants in order to find the optimal hardware design. This compute intensive approach explores many combinations of architectural parameters, such as pipeline depth, creating

variants for different resource budgets [16]. Unfortunately, building systems with multiple accelerators designed in this way leads to several issues. First, selecting between the many variants of each accelerator is computationally intensive. Second, this process does not easily permit accelerators to share common resources, which may duplicate underutilized logic. Third, there is no way for accelerators to dynamically grow or shrink if resource budgets change. An alternative is to design all accelerators into the system as one combined accelerator, but this approach can further extend design time and is inflexible.

ShrinkFit, an extensible framework that facilitates the design of multi-accelerator systems, is an answer to these challenges. This framework not only applies to the RoboBee application, but to hybrid systems in general. By relying on virtualization [49], ShrinkFit allows systems to grow or shrink accelerators to flexibly fit within shared FPGA resource budgets. It reduces the computational complexity of allocating resources to each accelerator, allows accelerators to share resources, and could be combined with dynamic reprogramming to support dynamic resizing. The implementation is based on an accelerator store [50], a memory resource for accelerators to save and exchange data. To support ShrinkFit, three new elements have been added: new features to the accelerator store, a "slicer" component for managing data transfers between ShrinkFit accelerators, and a wrapper interface that simplifies adding ShrinkFit capabilities to existing accelerators. These capabilities rely on hard logic blocks, dedicated to ShrinkFit and assumed to be built into the FPGA, in the same way FPGAs currently contain dedicated RAM, DSP, and general purpose hard cores. This chapter also introduces a software interface for building applications with ShrinkFit accelerators, including a software development kit (SDK) developed for the Python programming language which can be easily ported to other languages.

As the name suggests, ShrinkFit enables accelerators to fit within small FPGA budgets when necessary, and expand to increased resources for additional performance. This capability is demonstrated with four ShrinkFit accelerators developed for the bee brain prototype. Experimental results show that ShrinkFit enables performance of individual accelerators and the bee application to scale linearly with available FPGA resources. Overheads are low as well: ShrinkFit hard logic block overheads require less than 2% of overall FPGA die area, FPGA resource overheads range from 0%-8%, and application performance overheads are under 10% on average.

4.1 Motivation

The RoboBees project, a large collaboration between many research groups, seeks to build a swarm of bee-sized, flying robots. Each RoboBee, due to its small size and limits of wing-flapping lift, must be as light and low-power as possible. Yet, each RoboBee must perform complex visual and control algorithms, which require significant processing capability. To minimize power consumption while maximizing performance, the RoboBee's brain uses hardware accelerators as well as a general purpose Cortex-M0 processor. Unfortunately, to know which accelerators are needed and the performance requirements for each, it was necessary to first test designs on a small flying vehicle in order to explore the design space.

To evaluate accelerator needs, a custom designed helicopter brain prototype (HBP) circuit board was created that snaps onto a small helicopter (Figure 4.1). The HBP contains a low power Spartan-6 SLX150-1L FPGA, which offers the maximum resources within the helicopter's battery power and weight budgets. Including the FPGA chip, circuit board, and supporting components, the board weighs 4.2g, sufficiently below the 4.5g maximum for stable flight. The HBP also contains connectors to control the helicopter and to receive images from a low-resolution, high-frame rate camera optimized for autonomous flight [1]. The HBP therefore allows the RoboBees team to try new combinations of accelerators and see which can support autonomous flight.

To date, four accelerators have been identified for the project: image sharpening, edge detection, optical flow, and discrete cosine transform (DCT). The HBP uses all four accelerators simultaneously to process images from the on-board camera, and each image is processed by three tracks (Figure 4.2). The edge detection accelerator generates outlines for object recognition. At the same time, the image sharpening accelerator counteracts camera blur, and its outputs feed into the optical flow accelerator to estimate movement. The result, a two dimensional vector, allows the bee to avoid collisions and estimate distance flown. Finally, the DCT accelerator processes the image as part of a custom image compression algorithm based on the JPEG format, which compresses each image by 40% to 70%. The compressed image is then saved to flash memory for offline debugging.

Logic for each of the accelerators is implemented using the Vivado C-to-RTL high level language (HLL) compiler [4] to design each accelerator quickly. The alternative, manually implement-

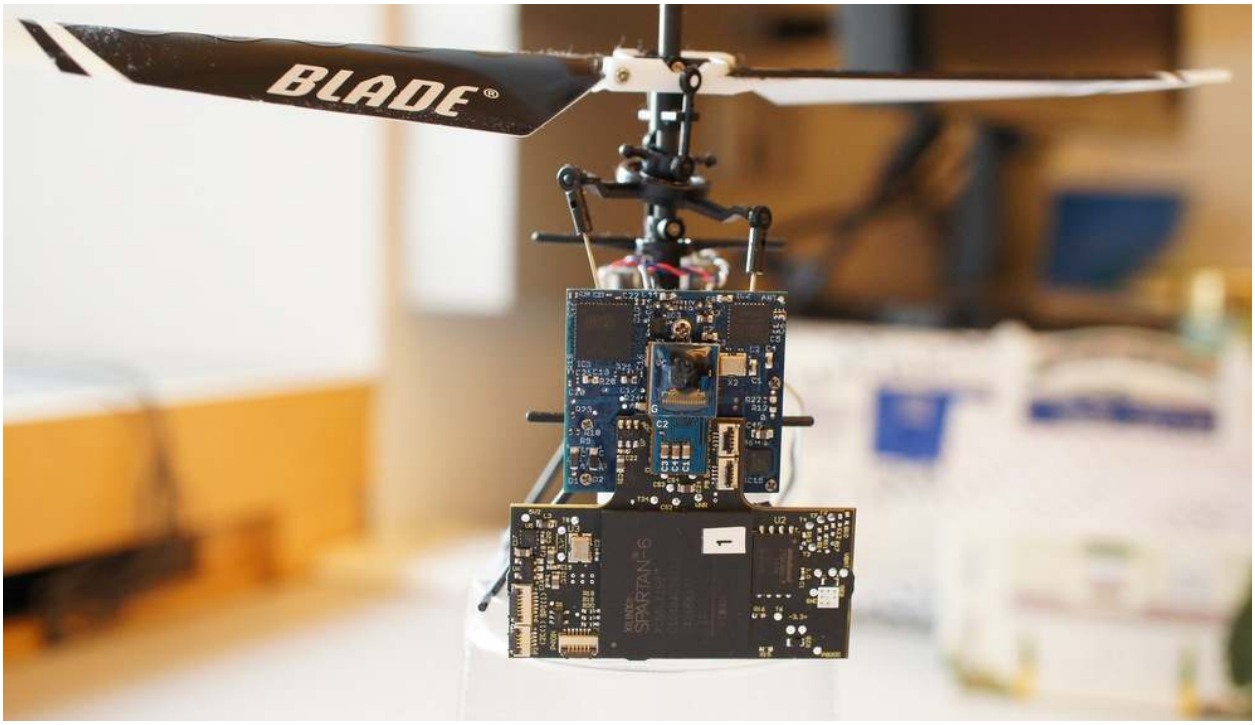


Figure 4.1: RoboBee Brain FPGA prototype
The helicopter brain prototype (black circuit board) is attached to a small helicopter and camera.

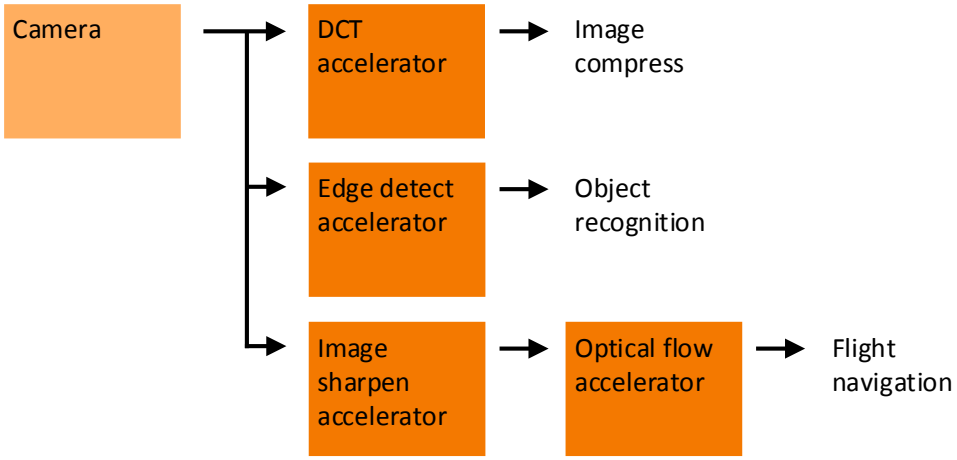


Figure 4.2: RoboBee application accelerators
The RoboBee application utilizes four accelerators. The brain prototype will use the results to compress images for offline analysis, recognize important objects, and avoid obstacles during flight.

ing each accelerator in Verilog RTL, typically takes more time. More accelerators will be added to the RoboBee brain as other project teams identify additional algorithms to accelerate, and will be

quickly ported to hardware using HLL compilers.

In order to determine FPGA resource requirements, it was necessary to first build the HBP to test different accelerator configurations. This situation motivated the development of the ShrinkFit framework, which allows the team to take full advantage of the FPGA's resources whether using a few accelerators early in development, or after adding more over the course of the project. Through this real-world design example, ShrinkFit was designed to generally address any FPGA system containing multiple accelerators, including hybrid processors. Throughout the remainder of this chapter, RoboBee accelerators and the RoboBee application are used to explain how ShrinkFit works and evaluate how it performs.

4.2 Conceptual approach

To support ShrinkFit, designers decompose accelerators into smaller, reusable "modules" of logic. Once decomposed, these modules can be combined to implement resizable accelerators via virtualization, and can be shared between accelerators.

4.2.1 Decomposition

ShrinkFit facilitates accelerator designs that are composed of smaller, reusable logic modules. To add ShrinkFit support to an accelerator, it must be decomposed into these reusable logic modules. Depending on the accelerator, different approaches may work best: some accelerators can be decomposed by pipeline stage, others by portions of each entry in a dataset (such as an image or matrix), or perhaps by subsequent entries in a dataset. For example, the image sharpening accelerator, used by the RoboBee brain, could be made up of multiple identical modules that each sharpens one region of an image. Assuming each image has sixteen regions, the accelerator requires sixteen sets of computations. One option would be to program all sixteen modules into the FPGA, each computing in parallel. In contrast, ShrinkFit enables the designer to resize the accelerator at any time by programming anywhere from one to sixteen modules into the FPGA, depending on performance and/or resource constraints. This resizability can shrink the image sharpen accelerator by 94% when only one module is used, but with at least a 16× increase in computation time. Ideally, this relationship between performance and resource utilization should be linear, i.e.,

doubling resources leads to double the performance.

The key concept behind ShrinkFit is that work performed by each module can be done without programming each module into the FPGA. Rather, the modules programmed into the FPGA need to be capable of doing the work of each module from the original design. To formalize the process of decomposing accelerators and designing resizable accelerators, ShrinkFit defines four terms:

- **Virtual modules (VMs)** are the modules from an accelerator's original design. In the case of image sharpen, which has sixteen regions, the accelerator always contains sixteen virtual modules. VMs represent the work to be done, rather than the logic doing it.
- **Physical modules (PMs)** are the actual logic blocks programmed into the FPGA. As few as one physical module can be programmed into the FPGA. More PMs can be added, but the number of PMs can never exceed the number of VMs. Programming fewer reduces FPGA resource utilization. Programming more PMs increases performance.
- **Module designs** refer to the algorithm a PM or VM implements. PMs of the same module, such as DCT, use the same RTL and are identical, whereas PMs of different module designs are not interchangeable. A convolution PM cannot do the work of a DCT PM.
- **Module contexts** contain the information a PM needs to act as a VM. Section 4.2.3 describes contexts in detail.

4.2.2 Building ShrinkFit accelerators with VMs

ShrinkFit accelerators consist of VMs of one or more module designs.

Image sharpening requires convolution, so the **image sharpen accelerator** uses sixteen VMs of a convolution module design. Each VM sharpens one of sixteen regions of the image, and when all VMs complete, the entire image is sharpened.

The **DCT accelerator** also uses sixteen VMs, but each is of a DCT module design, distinct from convolution VMs. Each DCT VM calculates frequency responses for one region of the image.

The **edge detect accelerator** contains sixteen convolution VMs, reusing the module design developed for the image sharpen accelerator. Edge detect adds sixteen magnitude VMs as well.

When using convolution VMs for edge detect, they produce two images, one for edges in the x-dimension, and another for the y-dimension. The magnitude VMs each process one of sixteen regions from both images, to create a third image that incorporates edges from both dimensions.

The **optical flow (OF) accelerator** uses two new module designs, OF region and OF merge. Sixteen OF region VMs estimate camera movement between the same region in two images. Then, the OF accelerator uses one OF merge VM to combine the data generated by each OF region VM into a single (x, y) vector.

4.2.3 Module contexts

In all ShrinkFit accelerators, including the four described above, VMs are an abstraction representing the work to be done. It is the PM, logic programmed into the FPGA fabric, that performs the computations. To bridge the gap between work and logic, PMs use "contexts," which are blobs of data that instruct a PM how to act as a VM. Each VM has a corresponding context, and by loading it, a PM can act as its corresponding VM. For example, a convolution context contains the image region its VM corresponds to. When a convolution PM loads a context, it immediately knows which region of each image to process.

Because there may be fewer PMs programmed into the FPGA than VMs in the accelerator design, these PMs must routinely switch to perform the computation of different VMs within the accelerator. This process is known as "context switching," because a PM will do the work of one VM for a short period of time, then switch contexts to do the work of another VM. This approach ensures that the work of all VMs will be completed regularly no matter how many PMs are programmed into the FPGA.

4.2.4 Accelerator resource sharing

Because PMs do not necessarily belong to one accelerator or another, they can be shared between accelerators. For example, image sharpen and edge detect accelerators both use sixteen convolution VMs. If a system used both accelerators simultaneously, each convolution VM would have a corresponding context, resulting in a total of 32 convolution contexts. And because the system contains 32 convolution VMs, one to 32 convolution PMs could be programmed into the FPGA. However, if fewer PMs exist in order to reduce FPGA resource utilization, they would all take

turns context switching into all 32 contexts, and do the work of all 32 convolution VMs in both accelerators. Rather than allocating different PMs to each accelerator, the accelerators share all PMs.

In rare cases, a system designer may wish to dedicate certain PMs to a single accelerator, rather than sharing PMs between accelerators. This is easily accomplished by creating two sets of contexts for the module design, and mapping some PMs to one set, and the remaining PMs to the other set.

4.2.5 Dynamic accelerator resizing

ShrinkFit could be combined with dynamic reprogramming techniques [71, 40, 53] to add support for dynamic accelerator resizing. PMs can be turned on or off individually, so turning one off would not disturb other PMs. With this in mind, accelerators could be resized while running and without interrupting computation by using dynamic reconfiguration techniques to add or remove individual PMs. Using dynamic accelerator resizing, ShrinkFit could continuously tailor FPGA resources to support workloads with fluctuating computational requirements and maximize performance.

4.3 Framework implementation

There are many ways to implement the conceptual approach described in the previous section as long as the implementation supports many PMs, enables PMs to context switch rapidly, and delivers PM input and output datasets quickly. With these requirements in mind, this section presents one implementation of the ShrinkFit concept.

Systems utilizing the ShrinkFit architecture include a general purpose core, several PMs, and ShrinkFit's hard logic blocks. Figure 4.3 provides a detailed illustration of how to use ShrinkFit for the RoboBees application. After PMs are first programmed into the FPGA fabric, the general purpose core configures PMs and various ShrinkFit hard logic blocks using the system bus to perform reads and writes as if the general purpose core was reading and writing to memory. These ShrinkFit hard logic blocks include: the accelerator store for maintaining VM contexts and data I/O, a slicer for tracking data transfers between VMs, and ShrinkFit wrappers to simplify the process of adding ShrinkFit features to existing accelerators. Since these logic blocks enable ShrinkFit features for

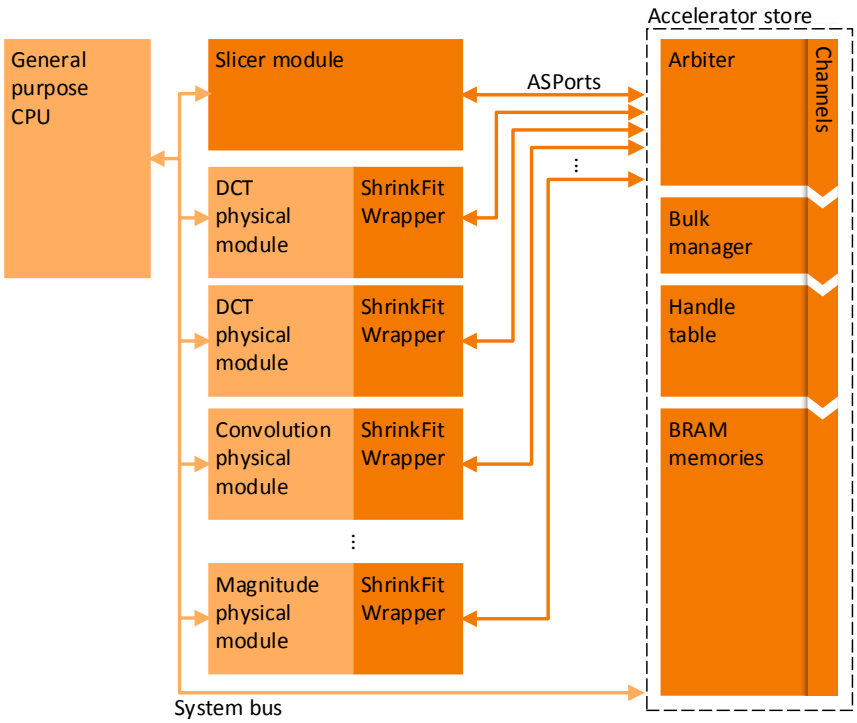


Figure 4.3: ShrinkFit framework architecture

The ShrinkFit framework consists of the accelerator store, slicer, and ShrinkFit wrappers. These components, darkened above, are permanently fabricated in the FPGA as hard logic blocks to obtain low die area overheads. All physical modules are programmed into the FPGA fabric as soft logic blocks. One or more general purpose cores may be implemented as hard or soft logic blocks.

any accelerator, and are not limited to specific accelerators, they can be hard coded into the FPGA chip to minimize resource overheads.

4.3.1 Accelerator store

To facilitate data storage and movement between PMs, the system employs the accelerator store [50], presented in Chapter 3. With some modifications, the accelerator store can be used to manage contexts and relay data between PMs. For maximum performance, accelerators often require fast access to input and output datasets. With this in mind, the accelerator store maintains direct connections with every PM programmed on the FPGA via ASPorts (Figure 4.3) to ensure low-latency communication.

Handles

To support ShrinkFit, two extensions were added to RA handles: multi-word transfers and a swap operation.

Multi-word transfers reduce arbitration delays when loading or storing large blocks of data. With a single ASPort request, VMs can load entire image regions. Multi-word transfers can access continuous blocks of data by incrementing the address after each word, or the VM can specify a custom access pattern by specifying a different address at each cycle.

Atomic swap operations ensure each context is held by no more than one PM, preventing data corruption. The swap operation, which simultaneously reads from and writes to an address, also prevents other ASPorts from accessing the same location. Using swap, a PM can ensure that no other PMs are using the same context.

Bandwidth and arbitration

The accelerator store supports systems containing many PMs and each may make requests simultaneously. The AS has an arbiter to satisfy as many PM requests as possible, and reject any remaining requests. The arbiter now uses a round-robin scheme to prevent starvation. Each PM's direct connection to the AS, called an ASPort, contains a signal that indicates whether the request was accepted or rejected. Each ASPort has an associated priority which determines the order that requests are accepted or subsequently rejected. If the request is rejected, the PM can try the request again on the next or later cycle. Rotating the priorities ensures that each ASPort can regularly access the AS.

AS bandwidth is represented in terms of "channels," which indicates the number of requests it can satisfy per cycle. Although the channel count is parameterized for any value, in practice, provisioning more than three channels had negligible performance benefits.

4.3.2 Slicer module

The ShrinkFit framework includes a *slicer* module that interacts with the AS and PMs and ensures all VMs can properly access input data or store output data in RA handles. For example, the edge detect accelerator contains sixteen convolution VMs, which output images to sixteen magnitude

VMs. To function correctly, all sixteen convolution VMs must each produce their regions of the output images before the magnitude VMs can consume them. In addition, the magnitude VMs must all consume these images before the convolution VMs can overwrite them with new output images. Because there are multiple VMs producing data and a different set of VMs consuming data, a FIFO handle would not suffice: as soon as one VM performed a get, the data would be lost to the other VMs. Instead, the slicer assists the VMs in ensuring that all VMs can reliably produce and consume data from RA handles.

To improve performance, the slicer supports varying amounts of buffering. Buffering is quantified in "buffer slots," which measure the number of entries that can be stored in the RA handle. For example, a handle with room for four images is sized to four buffer slots. By increasing the size of a handle to buffer more slots, a PM can batch more computation and improve performance.

Each buffer slot is in one of two states: produce or consume. When producing, one or more VMs write portions of the buffer slot. When all VMs finish producing, the buffer slot switches to the consume state. During the consuming phase, VMs read from the buffer slot. When all VMs finish consuming the buffer slot, it switches back to the produce state, and the slicer notifies VMs to produce the next buffer slot. In the edge detect accelerator, the buffer slot starts in the produce state as convolution VMs produce the x-direction and y-direction images. Once all sixteen VMs have finished producing, the slicer switches the buffer slot into consume mode, notifying the magnitude accelerator to begin consuming the images. And once all sixteen magnitude VMs have completed consuming, the slicer switches the buffer slot back to produce mode so that the convolution modules can store their next output images.

If the handles are sized for multiple buffer slots, VMs can produce and consume simultaneously from the handle (but not the same buffer slot). For the edge detect accelerator, if the convolution and magnitude VMs are connected by handles with two buffer slots, one buffer slot may be in a produce state and the other in a consume state. This allows magnitude VMs to consume an image generated by the convolution VMs, while the convolution modules simultaneously produce the next image into the other buffer slot.

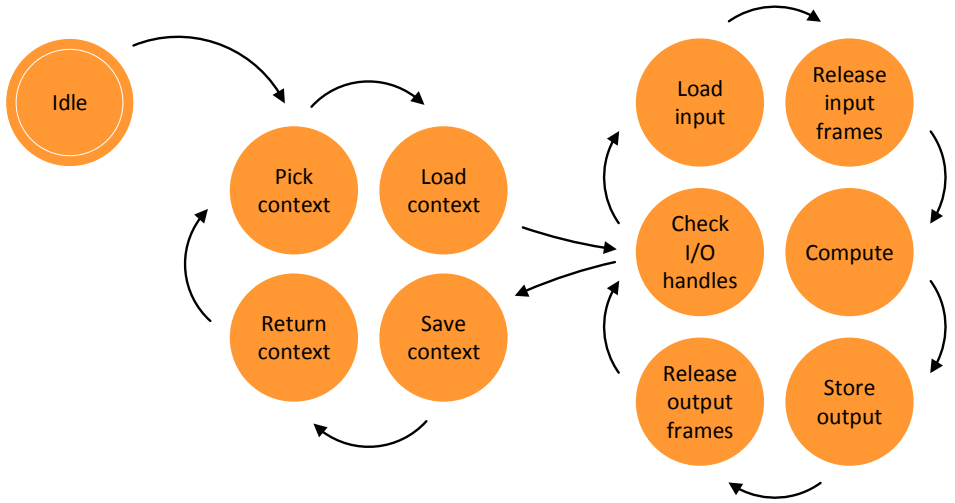


Figure 4.4: ShrinkFit wrapper state machine
Each ShrinkFit wrapper implements this state machine to context switch, load inputs, trigger computation, and store resulting outputs. Wrappers initially enter the idle state on startup.

4.3.3 ShrinkFit wrapper

The ShrinkFit wrapper is a small hard logic block that connects a PM to the accelerator store's ASPort. The wrapper implements common ShrinkFit tasks, including context switching, loading inputs, storing outputs, and interacting with the slicer (Figure 4.4). This logic was initially implemented within each PM, but resulted in duplicate logic in all PMs. Hence, the common logic was refactored into a generalized hard logic block.

ShrinkFit wrapper contexts

The wrapper defines a common context handle structure (Figure 4.5). Each module design has its own RA context handle, with a context for each of its VMs. The first words in the context handle each correspond to a different VM, forming a context directory. Each of these directory entries contain the location of the VM's context data and the length of the data. This approach was chosen to support variable-length contexts, rather than hard coding context data lengths to a set size.

This context directory approach allows wrappers to quickly check if a context is claimed by another VM. A slicer tries to claim a handle by swapping the context's directory entry with an invalid entry. If the wrapper receives an invalid entry back, it knows another PM has already

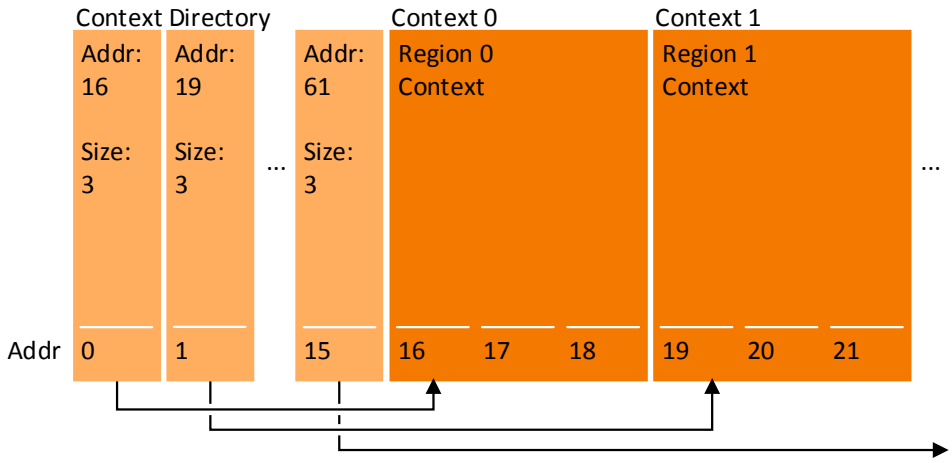


Figure 4.5: ShrinkFit wrapper context handle structure
ShrinkFit wrapper context handles contain a context for each VM. A context directory at the start of the handle identifies the locations of each context.

claimed this context, and tries to claim the next directory entry. If the PM receives any other value back, it knows it has successfully claimed the context. The wrapper then uses information in the directory entry to load the context. When the PM is done using the context, it saves any changes back to the context data, then writes the original directory entry back to the context directory.

ShrinkFit wrapper input/output

After loading the context, the ShrinkFit wrapper first checks if input handles have enough data to consume, and if output handles have enough space to produce the results into. If checks fail, the context is returned and a new one is chosen. Otherwise, the wrapper begins loading from input handles.

When loading input data, the wrapper's connection contains a RAM interface (enable, write, data in, address) to write the data to the PM. This allows the PM to store input data automatically, without programming additional control logic into the FPGA. The PM can also elect to manually load input data if it has special requirements.

After input loading completes, the wrapper prompts the VM to begin computation. When the VM indicates computation completes, the wrapper uses a process similar to loading inputs in order to store the computation outputs.

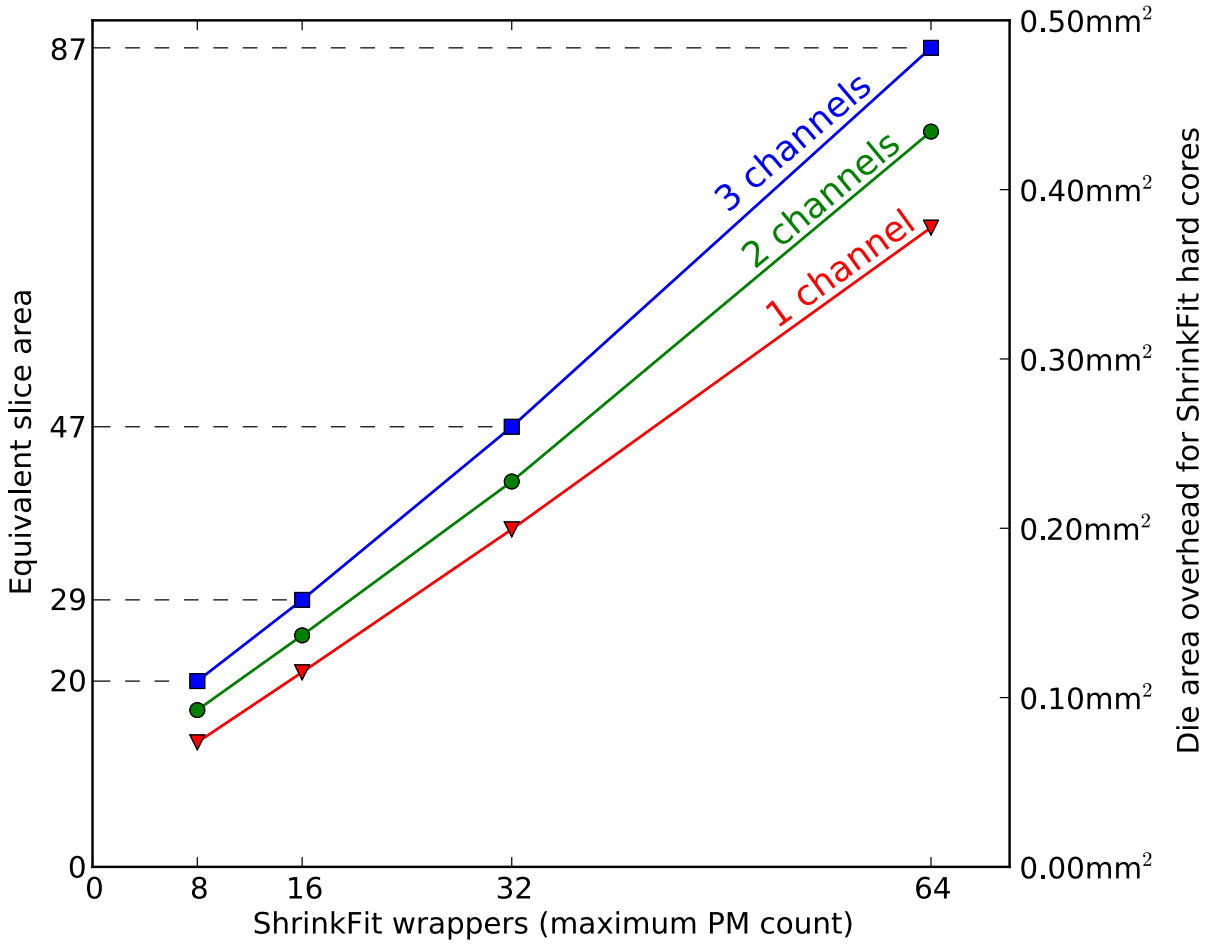


Figure 4.6: ShrinkFit hard logic block die area overheads
ShrinkFit hard logic block die area overheads are low for systems with many accelerators and well-provisioned bandwidth. Hard logic blocks are synthesized for a commercial 40nm process technology, and area is expressed in mm^2 as well as normalized to the area of a Spartan-6 FPGA slice.

4.3.4 ShrinkFit framework area costs

Because the ShrinkFit framework is generalized, rather than designed for a specific set of accelerators, it makes sense to hard code it into the die, rather than programming it into the FPGA. ShrinkFit also provides several opportunities for PMs to override automatic features if they desire a custom solution.

By utilizing hard logic blocks, the ShrinkFit framework has low area overhead. All three hard logic blocks—AS, slicer modules, and ShrinkFit wrapper—are synthesized using Design Compiler

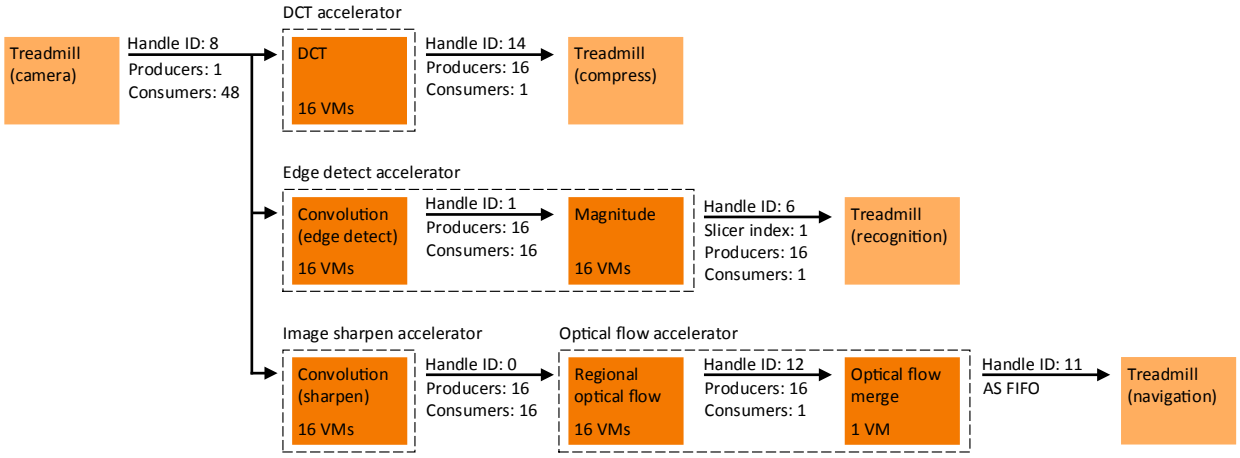


Figure 4.7: RoboBee application decomposed into VMs
Prior to implementing software, each ShrinkFit accelerator in the RoboBee application is decomposed into VMs and handles

D-2010.03 for a commercial 40nm process. Assuming a three-channel AS with ASPorts for 64 PMs, which is more than sufficient to maximize RoboBee application performance (only 36 are needed), Figure 4.6 plots the die area overhead versus the number of ShrinkFit wrappers. For comparison to reconfigurable resources, a commercial 40nm memory compiler was used to find the area of an SRAM equivalent to the 32x512, dual-port BRAMs found in the Spartan-6 ($0.0276mm^2$). This value is used to express the hard logic block area in terms of slices, the basic building block of FPGAs (this calculation is described in more detail in Section 4.5.1). Using the equivalent slice area in Figure 4.6, Section 4.6.4 later shows that the ShrinkFit hard logic block die will consume less than 2% of the FPGA's die area in both small and large FPGAs.

4.4 Software Development

Developing applications requires the system designer to perform two tasks: decomposing accelerators and configuring the ShrinkFit hard logic blocks. To simplify the latter step, a software development kit (SDK) called shrinklib was created for the Python programming language.

4.4.1 Decomposing accelerators

Before any software can be implemented, the accelerators in the application design (Figure 4.2) must be decomposed into their corresponding VMs (Figure 4.7). For example, the edge detect accelerator decomposes into sixteen convolution VMs and sixteen magnitude VMs. This decomposition will also include handles to represent data connections, such as the images produced by convolution VMs and consumed by the magnitude VMs.

The application designer must decide how many buffer slots to allocate to each handle and calculate how many VMs produce and consume from each handle. Contexts for each module design are stored in RA handles, so handles must be allocated to store contexts as well.

4.4.2 Configure ShrinkFit hard logic blocks

Once the application designer decides on the configuration of VMs and handles, it is time to implement software. The software program uses system bus reads and writes to configure the ShrinkFit hard logic blocks, just as it would to read from or write to memory. The program must configure a few things:

1. Create handles (to connect VMs and store contexts)
2. Configure the slicer with the number of VMs that produce and consume each handle
3. Store contexts for each VM in context handles
4. Configure each PM's ShrinkFit wrapper
5. Start each PM's ShrinkFit wrapper

All of these routines consist of multiple reads and writes to the system bus.

4.4.3 Shrinklib SDK

The shrinklib SDK includes routines to automate the application development steps after decomposition. Although shrinklib is currently implemented in the Python programming language, it simply performs system bus reads and writes, and is easily ported to other languages. The RoboBee application demonstrates how shrinklib is used below:

Using shrinklib, the application first initializes the ShrinkFit framework and creates handles (step 1). Ellipses are substituted in place of repetitive code:

```
brain.SetUpBrain()
camera_image_handle = shrinklib.AsHandle(
    spi, hid=8, start_addr=0x00018000, word_count=4096)
...
dct_ctxt_handle = shrinklib.AsHandle(
    spi, hid=13, start_addr=0x0001F800, word_count=512)
dct_coefs_handle = shrinklib.AsHandle(
    spi, hid=14, start_addr=0x00000000, word_count=8192)
handle_table = shrinklib.AsHandleTable([
    camera_image_handle, ..., dct_ctxt_handle, dct_coefs_handle])
handle_table.CommitToAs()
```

The RoboBee application uses a similar set of calls to map producer and consumer counts to each handle (step 2). Afterwords, the application builds each module design's context handle, configures each ShrinkFit wrapper, and starts the PMs (steps 3-5). For example, this code performs steps 3-5 for all DCT PMs:

```
virt_dct2_set = shrinklib.VirtDct2Set(
  spi=spi, slicer=slicer, physical_module_count=dct_pm_count,
  context_handle=dct_ctxt_handle)
virt_dct2_set.AddContexts(
  in_handle=camera_image_handle, out_handle=dct_coefs_handle)
virt_dct2_set.CommitInit()
virt_dct2_set.StartPhysicalModules()
```

The code used to initialize the other four PM designs is almost identical.

If new PM designs are created, adding support for them to shrinklib is straightforward. The module designer only needs to write two methods for the new design. The first routine uses the system bus to configure each of the ShrinkFit wrappers for the module design, the second builds the context handle for each module design.

4.5 ShrinkFit module evaluation

PMs are the building blocks of all ShrinkFit accelerators, and by extension, the applications that use them. Before evaluating the RoboBee application and the four ShrinkFit accelerators it uses, the five PMs they are built from—convolution, magnitude, DCT, OF region, and OF merge—are

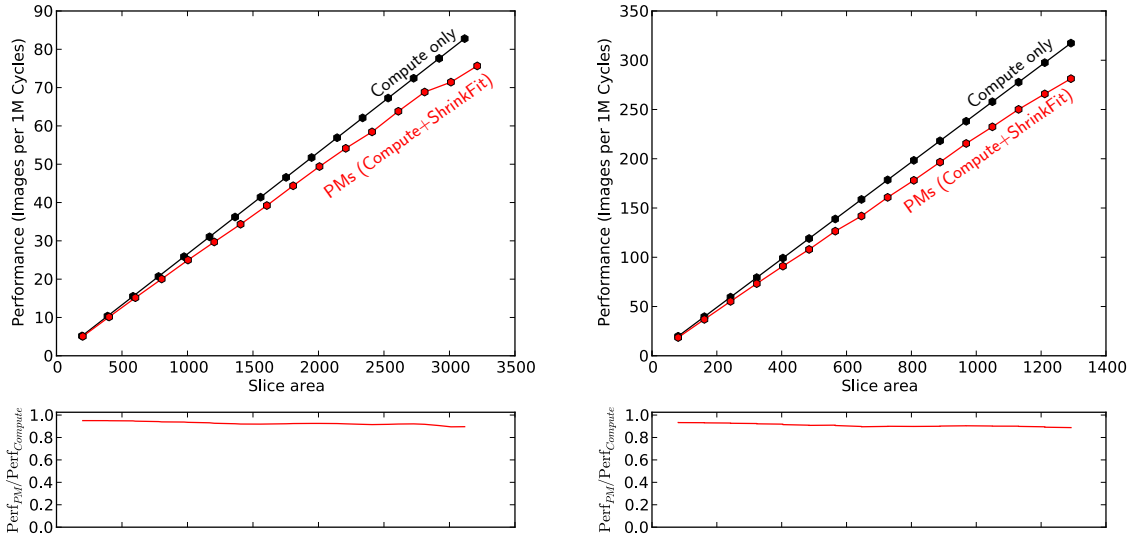
considered. Specifically, two aspects of the PMs are evaluated. First, performance scales up linearly with FPGA resources as more PMs are added. Second, regardless of whether a few or many PMs are programmed into the FPGA, performance overheads and resource overheads remain low.

4.5.1 ShrinkFit PM implementations

For each of the five RoboBee PM implementations, compute logic and ShrinkFit functionality is partitioned into separate blocks. Compute logic, designed using the Vivado C-to-RTL compiler [4], only contains the logic required to perform the PM's computation and BRAM memories to hold input and output data. In other words, the compute logic blocks do not contain any optimizations for ShrinkFit interfacing or functionality. To add ShrinkFit support, each PM has additional "glue logic" to connect the compute logic block to a ShrinkFit wrapper. Since the wrapper performs most of the tasks related to ShrinkFit, the glue logic simply connects the wrapper to the corresponding PM and takes care of any special cases. For example, the convolution module may produce outputs to one handle if performing image sharpening, or two handles if performing edge detect. Convolution glue logic guides the wrapper to accommodate this choice properly.

In all five RoboBee PM designs, glue logic resource requirements are small compared to their corresponding compute logic blocks, between 0.0% and 7.8% (Table 4.1). These low resource overheads are largely thanks to the ShrinkFit wrapper, which implements the majority of ShrinkFit functionality as a hard logic block. In order to compare the glue logic and compute logic resource costs, each PM is synthesized using Xilinx ISE 14.4 for the Spartan-6 FPGA used in the HBP, with and without glue logic. Like most FPGAs, the Spartan-6 contains four basic primitives: registers and lookup tables (LUTs), DSP blocks optimized for addition and multiplication, and BRAM blocks for efficiently storing large datasets. Registers and LUTs are contained within slices, of which hundreds if not thousands exist in the FPGA.

In addition to counting these primitives (register, LUT, DSP, and BRAM) in each design, a "slice area" resource cost is calculated combining primitives into a single metric. This metric packs registers and LUTs into slices (16 registers and 8 LUTs per slice) to obtain a slice count, and determines the die area of DSPs and BRAMs relative to the area of a slice (DSPs and BRAMs consume the area of 4.95 slices, according to PlanAhead floor plans). Total slice area is the sum of slices used by registers and LUTs, combined with the relative slice area of DSPs and BRAMs.



(a) Convolution PMs

(b) Magnitude PMs

(c) DCT PMs

(d) OF region PMs

Figure 4.8: Single PM design resource-to-performance trade-off

All four systems contain one to sixteen PMs of the same module design, demonstrating that ShrinkFit scales performance with FPGA resources, as well as low resource and performance overheads. Each configuration processes 100 images using a 3-channel AS. "Compute only" considers only the compute logic without ShrinkFit overheads. PM performance considers ShrinkFit glue logic resource and performance overheads, in addition to compute logic costs. Slice area considers the area consumed by registers, LUTs, DSPs, and BRAMs, relative to the area of a Spartan-6 slice. Secondary plots compare $Perf_{PM}/Perf_{Compute}$, the ratio between the two series.

Table 4.1: PM FPGA resource overheads

FPGA resource overheads (Δ) for all five module designs are low, ranging from none to +7.8%. "Slice area" equals the sum area of FPGA primitives (registers, LUTs, DSPs, and BRAMs) relative to the area of a Spartan-6 FPGA slice. Each slice contains 16 registers and 8 LUTs. DSPs and BRAMs are each approximately 4.95x the area of a slice.

	Convolution	*Magnitude*	*DCT*	*OF region*	*OF merge*
Slice Area					
Compute	195	81	138	120	261
PM	201	81	149	126	261
Δ	+3.0%	+0.0%	+7.8%	+5.0%	+0.0%
Registers					
Compute	463	92	409	178	772
PM	495	101	436	208	772
Δ	+6.5%	+8.9%	+6.1%	+14.4%	+0.0%
LUTs					
Compute	680	221	391	337	863
PM	702	221	436	361	864
Δ	+3.1%	+0.0%	+10.3%	+6.6%	+0.0%
DSPs					
Compute	2	2	4	5	4
PM	2	2	4	5	4
Δ	+0.0%	+0.0%	+0.0%	+0.0%	+0.0%
BRAMs					
Compute	3	3	4	2	5
PM	3	3	4	2	5
Δ	+0.0%	+0.0%	+0.0%	+0.0%	+0.0%

In the case of magnitude and OF merge modules, glue logic adds no overheads. This is due to underutilized slices in compute logic blocks, containing too few registers or LUTs to completely pack slices. The wrapper logic is able to utilize the unused primitives without adding additional slices, thus resulting in no additional overheads.

4.5.2 Evaluation methodology

To analyze scalable PM performance and overheads, four system design scenarios are analyzed, each limited to only use one out of the four module designs: DCT, convolution, magnitude, and

Table 4.2: PM compute logic block processing delays

Each compute logic block exhibits differing processing delay requirements, sometimes dependent on input data. Although these variable schedules could threaten FPGA resource/performance scaling, the ShrinkFit framework is not noticeably affected.

	Cycles per image region			
	Average	*Minimum*	*Maximum*	*Max-Min* Δ
Convolution (sharpen)	13,520	12,665	14,403	12.1%
Convolution (edge detect)	12,079	11,315	12,867	12.1%
Magnitude	3151	2600	3250	20.0%
DCT	33,061	33,061	33,061	0.0%
OF region	1251	1173	1331	11.9%
OF merge	530	530	530	0.0%

OF region. OF merge, the fifth module design, is not analyzed because only one can be used per optical flow accelerator. For each system, different PM counts are considered, ranging from the minimum (1) to the maximum (16). These systems also contain an accelerator store with three channels of bandwidth and four buffer slots per handle.

The performance analysis relies on ModelSim 10.1, which performs cycle-accurate simulation of all runs using synthesizable RTL for all hard logic blocks and PMs. Each system utilizes a "treadmill" testing module, ensuring repeatable and error-free tests. The treadmill module injects pre-recorded camera images into the system in the same order and verifies PM outputs using corresponding checksums. Performance is determined by measuring the number of cycles required to process 100 images. To evaluate ShrinkFit performance overheads, each PM's performance is compared with the compute logic block's performance. The subset of cycles spent utilizing the compute logic block is also recorded. This measurement reveals the theoretical maximum performance of the PM without any ShrinkFit overheads (Table **??**). Dividing actual ShrinkFit system performance by this theoretical maximum ($Perf_{PM}/Perf_{Compute}$) reveals the performance overhead of the ShrinkFit system.

Implementations of the ShrinkFit framework RTL and PM RTL have been verified to correctly synthesize and work in the HBP hardware. However, the HBP platform could not be used for thorough performance analysis. Because the off-the-shelf Spartan-6 FPGA does not include

ShrinkFit hard logic blocks, all ShrinkFit features were programmed in as soft logic blocks, consuming much more of the FPGA's resources. This limited the number of PMs that could be added and introduced additional overheads that would not occur with ShrinkFit hard logic blocks.

4.5.3 PM performance scalability

Since all ShrinkFit accelerators are built with PMs, it is important to ensure they work well individually within the framework before combining them to compose accelerators for different applications. Hence, the performance scalability of individual PM implementations is evaluated first. Figure 4.8 plots the performance versus resource utilization (slice area) for the four resizable types of PM designs. For each of the PMs, the plots show how performance scales for "Compute only" and "PMs." "Compute only" data points correspond to compute logic without any other resource or performance overheads. This is a highly optimistic upper bound which does not consider the costs of context switching, loading input data to process, or storing the resulting output data. However, many of these overheads would be present whether or not ShrinkFit is used. For example, inputs and outputs will always need to be loaded. The "PMs" data points include these performance and soft logic resource overheads in order to evaluate how PMs perform in an actual ShrinkFit system. Each data point in Figure 4.8 represents a system with a different number of PMs. The points at the far left represent a system with a single PM. Proceeding to the right, each consecutive point uses more FPGA resources to add an additional PM. This continues until reaching the rightmost point, representing a system utilizing a maximum 16 PMs. The corresponding $Perf_{PM}/Perf_{Compute}$ plots show how overheads scale again with respect to resource utilization for all four PMs.

The results demonstrate that all of the PMs successfully achieve a linear performance-to-resource relation, with some minor exceptions. For each PM added to the system, performance roughly increases by a nearly constant factor. As the number of PMs increase, the slightly lower slope can be attributed to contention in the accelerator store. The $Perf_{PM}/Perf_{Compute}$ plots better illustrate this trend. While a slight downward slope can be seen for all PMs, these plots verify that ShrinkFit overheads are consistently low even as more PMs are added. The larger overheads for OF region are due to large inputs (regions of an image) and short execution times (Table **??**). OF region spends a considerable portion of its time loading inputs, a delay that would occur even if system architectures other than ShrinkFit were to be used. For this reason, and considering that

the average $Perf_{PM}/Perf_{Compute}$ is still high at 78.96%, the PM implementation of OF region, and all other module designs, is sufficient for the goals of ShrinkFit.

4.6 RoboBee application evaluation

Given that PMs can scale performance with FPGA resources, and do so with low area and performance overheads, the case where PMs are combined to form four accelerators and implement the RoboBee application shown in Figure 4.7 is now considered. The system should continue to scale performance with FPGA resources while adding more PMs into the system. The following evaluation not only demonstrates this is achievable, but that overheads also remain low. Further, results demonstrate that less channel bandwidth is necessary to support the RoboBee application than some PMs require when considered individually. Finally, the role of buffering in regards to performance is investigated.

All runs of the RoboBee application follow the same testing approach as with single module evaluations. The treadmill module is used to inject test images and verify output checksums. For each test, performance is measured as the number of cycles required to completely process 100 camera images. All test results are obtained from cycle accurate RTL simulations. In addition, a system using one PM of each of the module designs is implemented and deployed to the RoboBee brain prototype FPGA. The application was successfully run on the FPGA prototype without error (on-FPGA and simulation results matched).

4.6.1 Application evaluation overview

ShrinkFit systems processing the RoboBee application are considered first. Like the previous figure, Figure 4.9 plots performance versus slice area for "Compute only" and three "PM" implementations with different AS bandwidth assumptions (1, 2, and 3 channels). Again, "Compute only" data points only include compute logic resource and performance costs and do not include ShrinkFit overheads due to context switching, loading input data, or storing output data. Each point in the plot represents a different set of PMs. Points on the left side represent the minimum set of PMs, one each of the five module designs (convolution, magnitude, DCT, OF region, and OF merge) required by the application. This configuration requires the fewest FPGA resources possible. Each

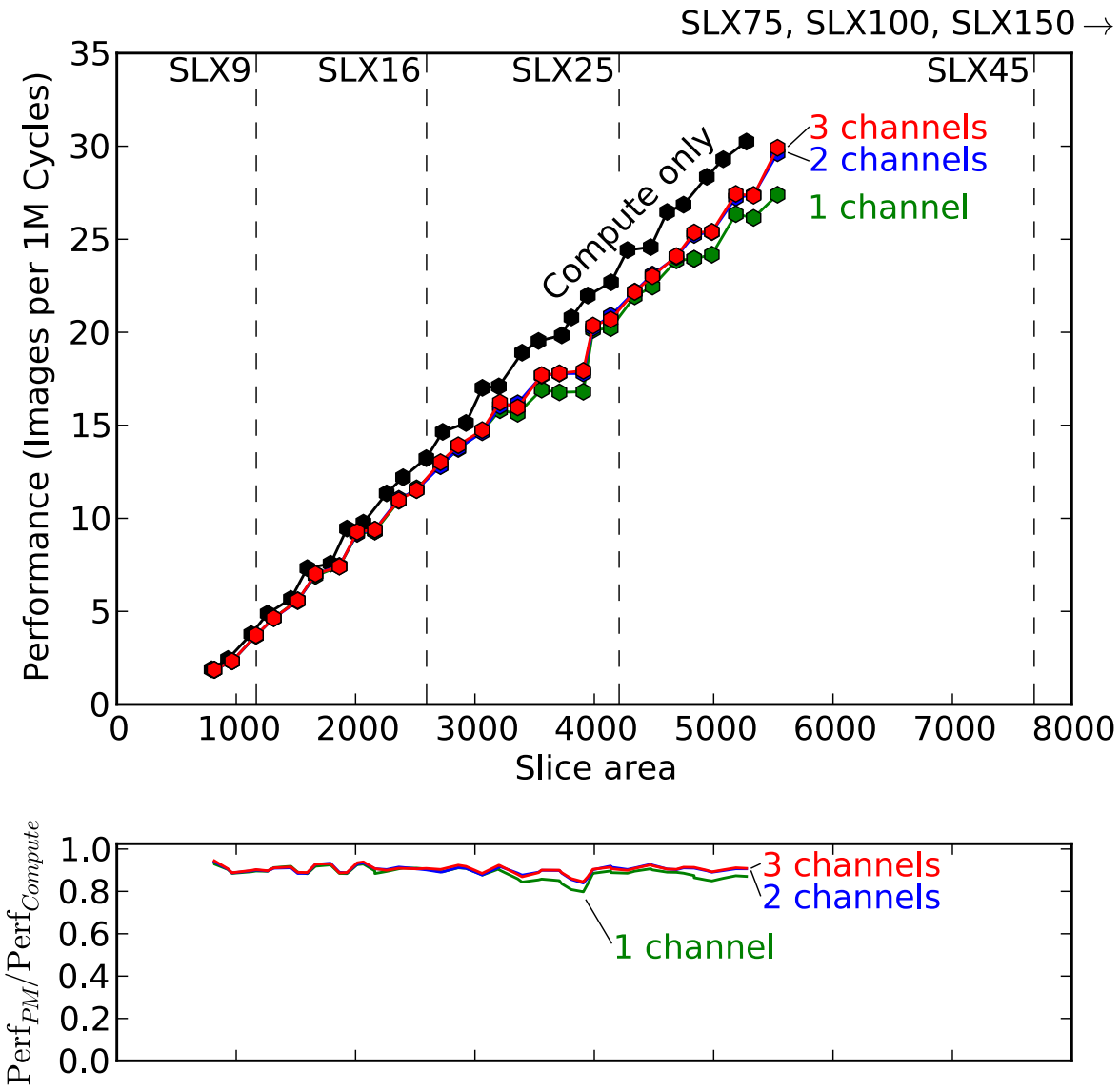


Figure 4.9: RoboBee application resource-to-performance trade-off bandwidth impact When running full applications, AS bandwidth needs are low: two channels of AS bandwidth is sufficient for the RoboBee application, and one channel performance is only slightly less. Systems with varying PM counts and one, two, or three channels of AS bandwidth are considered. FPGA resources for Spartan-6 FPGA models are noted at the top.

point to the right adds an additional PM to the system. Unlike the previous evaluations that considered a single module design, the system designer must decide which of the four PMs to add to the system (because only one OF merge VM is used for the optical flow accelerator, there is no benefit from adding additional OF merge PMs). To decide which of the four PMs to add,

each PM's compute-only average performance (Table **??**) is used as an approximation for the PM's performance and use these approximations to exhaustively calculate the expected performance of each possible system configuration. This estimation is close enough to make good decisions, due to the previous section's findings that PM performance overheads are low. From the roughly 130,000 possible PM permutations, the highest performing configurations were progressively chosen for each subsequent point with increasing slice area up to the number of PMs that maximize the overall performance of the application. This exhaustive search required less than one second on a typical desktop computer. More efficient search strategies are certainly possible, but the exhaustive approach performed well in practice for the workload.

The plot clearly shows that ShrinkFit again enables performance to scale up with increasing FPGA resource utilization. The somewhat jagged data points in the plot is an artifact of the different resource and performance characteristics of the PMs. Each module design's PM implementation consumes different FPGA resources and requires different amounts of time to complete their operation. In other words, the performance gained and resources consumed by adding a PM varies between module designs. In addition, applications may chain VMs in series, and adding an additional PM will only improve performance until it alleviates the critical path, shifting it to a different PM design. Although the point-to-point relation is jagged, the application's overall trend continues to be roughly linear.

These results verify that performance overheads are low when processing the application across the full range of slice area, as seen by the $Perf_{PM}/Perf_{Compute}$ plot. Although there is slightly more variance in this ratio than with single module evaluations, it is still relatively flat. In addition, the ratio is always high, on average at 90.34% for systems with a three channel AS, indicating performance overheads are low.

4.6.2 Bandwidth impact

Experimental results in Figure 4.9 additionally reveal that bandwidth has less of an effect on performance than for single module systems. Some of the module designs in the single module evaluation required an AS with three channels to achieve high performance. However, when considering the full application, three channels only improve performance over two channels by 0.20% on average, an insignificant difference. Even one channel performance is only 1.72% less on average than with

three channels.

Bandwidth needs for the application are lower than for single modules because needs are determined by the slowest modules, not the fastest. OF region PMs require up to three channels when large numbers of PMs are present since computation is so fast that loading input images (utilizing bandwidth) requires a significant portion of the PM's cycles. But because the OF region PMs are so fast, they are rarely on the critical path when slower PMs, such as DCT, are present. As a result, the application never needs to program enough OF region PMs to require three channels.

These results also demonstrate the efficacy of PM sharing between ShrinkFit accelerators. Both edge detect and image sharpen accelerators make use of convolution PMs. These PMs are not partitioned to one accelerator or the other, rather, all PMs rapidly switch between both accelerators. Further, the lowest resource configurations use a single convolution PM, shared by both accelerators. Despite switching between both accelerators, the system achieves high performance.

ShrinkFit quickly identifies when the system achieves maximum performance and adding additional PMs would waste resources. PM configurations that consume 16x more resources than the minimum sized configuration are not considered, because maximum performance is achieved well before this point. Like bandwidth, performance is dictated by the slowest module limiting the critical path. Therefore, when the slowest module programs its maximum number of PMs into the FPGA logic (sixteen DCT PMs in the RoboBees application) the critical path cannot be reduced by adding PMs from other module designs. Once the maximum number of PMs for a module design have been programmed into the FPGA, it is quickly apparent that there is no need to consume more FPGA resources with other PMs. In the case of the RoboBees application, this occurs with the following PM counts: 16 DCT PMs, 13 convolution PMs, 2 magnitude PMs, 1 OF region PM, and 1 OF merge PM. By quickly identifying this maximum performing configuration, ShrinkFit prevents programming additional, unnecessary logic.

4.6.3 Buffering impact

In contrast to bandwidth insensitivity, Figure 4.10 shows buffering has a significant effect on performance as PM count grows. For this experiment, systems with the same PM selections as in the bandwidth evaluation were used. However, instead of varying bandwidth, each handle is sized to hold either one, two, or four buffer slots. In systems with low PM counts, $Perf_{PM}/Perf_{Compute}$ re-

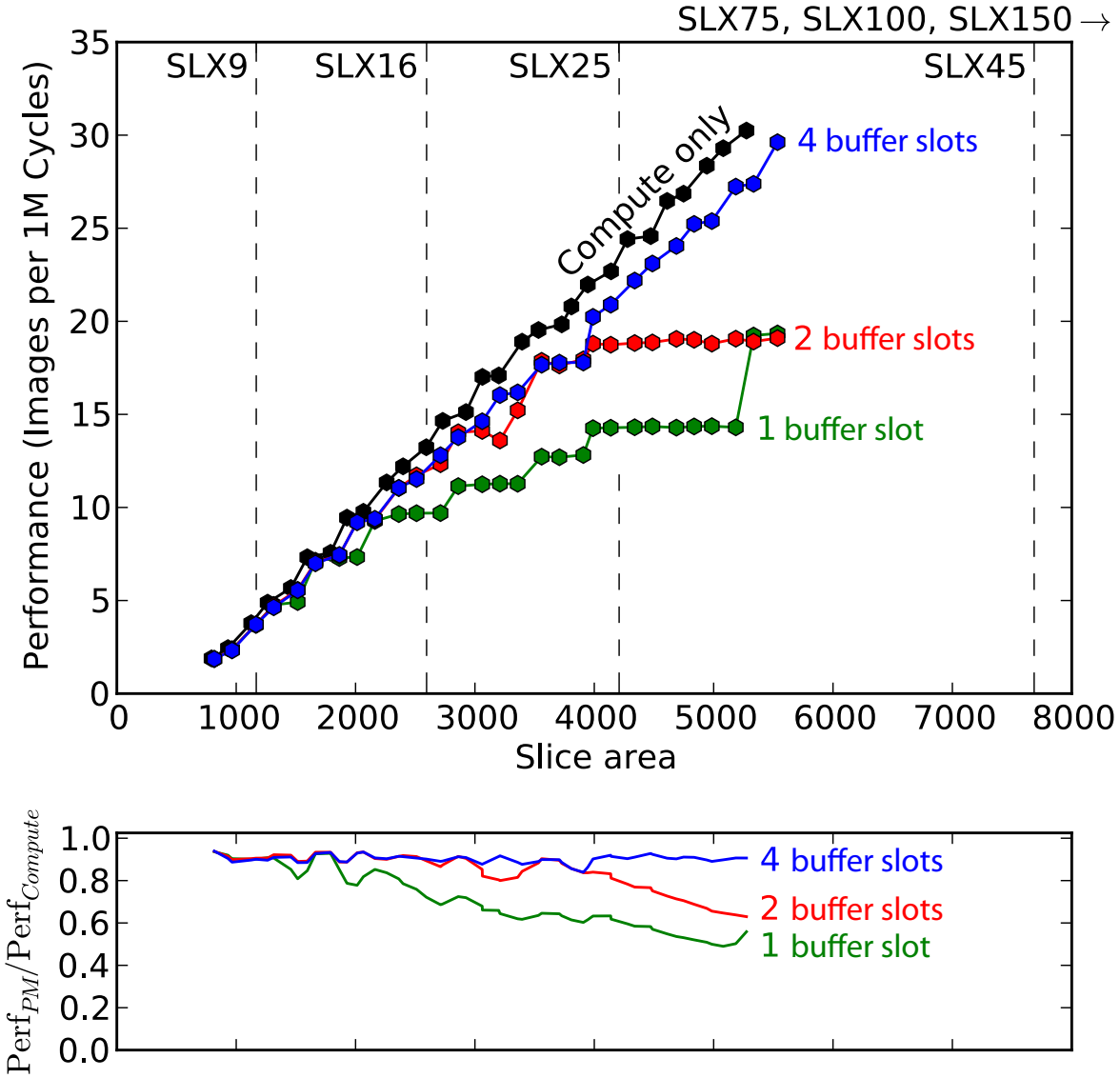


Figure 4.10: RoboBee application resource-to-performance trade-off buffering impact
Buffering is essential for high performance with many PMs: four buffer slots per handle is necessary to scale performance with upper PM counts. Systems with varying PM counts and one, two, or four buffer slots per handle are considered. FPGA resources for Spartan-6 FPGA models are noted at the top.

mains high regardless of buffer size, indicating low performance overheads. However, as PM counts rise, buffering less than four buffer slots constrains performance and lowers $Perf_{PM}/Perf_{Compute}$.

Buffer size has significant impact on performance due to pipelining effects between modules connected in series. For example, if a handle is sized for a single buffer slot, that buffer slot can only

be used to produce or consume at any given time. Therefore, for modules on the critical path, only half may be actively processing at any time. This effect is lessened for handles sized for two buffer slots, which allows connected VMs to produce and consume simultaneously. Still, it is unlikely that both VMs will complete producing and consuming at exactly the same time, and therefore one will stall while the other completes. Using handles with room for four buffer slots decouples modules in series, and improves pipelining performance accordingly. Because these limitations would apply to any accelerator based system, the experimental results show that buffering is important for any accelerator based architecture and does not apply solely to the ShrinkFit framework.

To obtain a direct comparison between "Compute only" and actual PM performance while investigating buffering, the resource overhead of using different levels of buffering is not considered in Figure 4.10. "Compute only" assumes an unlimited amount of buffering, and because any accelerator based system would require amounts of buffering at least equivalent to ShrinkFit, omitting the resource cost results in the fairest comparison. Using four buffer slots rather than one requires an additional area equivalent to 159 Spartan-6 slices, and is only necessary to increase buffering when larger PM counts are used (and more resources are available on the FPGA). In practice, the cost of adding additional buffering consumes a few percentage points of resources, and would apply equally whether or not ShrinkFit was used.

4.6.4 Hard logic block area overheads

It is not surprising that FPGAs with larger resources can fit more PMs: both Figure 4.9 and Figure 4.10 indicate the total slice area of reconfigurable logic that each Spartan-6 FPGA contains. Smaller FPGAs, such as the SLX9 can only fit systems with a few PMs, whereas the larger SLX75 can fit enough to obtain the maximum achievable performance [3]. As such, the SLX9's ShrinkFit framework would need to provision support for fewer PMs, which would reduce the percentage of FPGA die area consumed by ShrinkFit hard logic blocks. If Spartan-6 processors were to include ShrinkFit hard logic blocks, provisioning 16 PMs for the SLX9, 32 PMs for the SLX16 and SLX25, and 64 PMs for larger Spartan-6 FPGAs should be sufficient if not over-provisioned. For flexibility, more ShrinkFit wrappers are provided than used for the RoboBee application. For the SLX75 and up, almost half of the 64 wrappers are unused and available for future expansion. Using data from Figure 4.6, the equivalent slice area of ShrinkFit hard logic blocks is small by comparison. With

this provisioning scheme, ShrinkFit hard logic blocks require less than 2% of reconfigurable die area for small and large Spartan-6 FPGAs.

4.7 Related work

Several works target multiple resource budgets by creating variants of the same accelerator. Cong, et al., use the Vivado C-to-RTL compiler to survey architectural parameters to create accelerator variants for different resource budgets [17]. As discussed in this chapter, this approach leads to challenges when building multi-accelerator systems. Elastic computing manually designs multiple variants of the same accelerator using different algorithms [75]. This approach is difficult to scale as it depends on significant manual design and the existence of multiple implementations of the same algorithm.

Other works have investigated approaches to manage multiple accelerators in a single system. Dales, et al., introduced an approach for a hybrid FPGA+GP processor to switch between accelerators and software execution [21]. This work does not use accelerator variants or resize accelerators. FPMR adds hardware support for MapReduce algorithms to resize an accelerator [66]. This work is limited to the use of a single accelerator and algorithms which fit within MapReduce semantics. CHARM and DRP use generic hard compute logic blocks instead of FPGA slices to create reconfigurable accelerators [18, 31].

Previous works have used virtualization concepts to support hardware acceleration. Kalte, et al., use contexts to store accelerator state, but for the purpose of pausing or relocating accelerators in FPGA fabric rather than resizing them [40]. C-Cores introduced an approach for ASIC designs that automates software-hardware codesign using virtualization-like state management and is limited to single accelerators [73].

Chapter 5

Future directions

The accelerator store is developed to be an extensible, open platform, rather than a self-contained work. The ShrinkFit framework, built on top of the accelerator store, was also designed to be a part of the evolving needs of accelerator-based systems rather than a isolated work. As such, many opportunities exist to build on top of both works, and many avenues for research are possible. In this chapter, several research directions are highlighted.

5.1 Accelerator store scalability

The accelerator store was originally designed, for the purposes of Chapter 3, to interact with up to 16 accelerators. Beyond this point, the arbiter's latency becomes long because the computational complexity of arbitration is linear with regards to accelerator count. For many-accelerator systems, a distributed accelerator store (DAS) approach was proposed, using multiple accelerator stores connected by a grid OCN. ShrinkFit was able to increase the number of supported accelerators to 64 (and perhaps beyond) by taking advantage of the relative performance differences between hard logic blocks (ASIC) and FPGA logic. However, using the accelerator store with more than 16 accelerators per accelerator on an ASIC platform (rather than FPGAs) would require an implementation of the DAS design or a lower-latency arbiter (or both). The DAS approach is described in Section 3.2.3. To reduce the arbiter's latency, at least two options exist: subset arbitration and multistage arbitration.

5.1.1 Subset arbitration

Subset arbitration only considers a subset of accelerators at each cycle. For example, if an accelerator store serviced 64 accelerators, it might only consider a window of 16 accelerators at each cycle, and automatically reject the other 48 potential requests outside that window. Although this would increase the worst-case time for an AS request to be accepted by the AS, the subset approach is unlikely to increase delay by much in practice. First, bulk requests will not be affected once accepted, so multi-word requests will only have to pay the additional arbitration cost once per request rather than once per word. Second, AS contention already prevents accelerators from making AS requests on the first attempt in many cases. Accelerator requests outside the sliding window could be viewed as low-priority requests that would be unlikely to be accepted even if considered.

The window could be moved using existing priority rotation logic. To ensure that requests from all accelerators are considered regularly and prevent starvation, the window could be rotated by $size_{window} - 1$. Rotating by the window size, rather than by the current single increment, ensures that all accelerators are considered as regularly as possible. The -1 term is necessary to prevent starvation, otherwise the same accelerators would always be the lowest priority and could be starved.

To further reduce delays, each channel could also consider a different sliding window. For example, if a four channel AS was used in the previous example, each channel could consider one of the four sets of sixteen accelerators. In this fashion, each accelerator could be considered at every cycle.

5.1.2 Multistage arbitration

A related approach to reducing arbitration delay is to make arbitration a multi-cycle process, rather than the current single-cycle approach. The advantage to implementing multistage support is that available channels are never wasted. In the subset approach, if the accelerators in the sliding window did not fully utilize available AS channels, AS bandwidth that could be used by accelerators outside the window would be wasted. However, the multistage approach does add considerable complexity to both the AS implementation, as well as the accelerators. Multistage arbitration could be viewed as the subset approach, but with copies of the arbitration logic for each stage, and would increase

the area and power overheads for arbitration. In addition, accelerators would no longer know in the same cycle if requests were accepted or rejected, and therefore would require additional logic to handle this variable notification latency.

5.2 Unified system+AS memory

Currently, data stored in the AS is not directly accessible to the GP-CPUs. Rather, GP-CPUs must use a bridge accelerator to access data word-by-word or DMA data between the AS and system memory. To improve interaction between the GP-CPUs and accelerators, it would be preferable for AS memory to be accessible over the system bus directly, so that it does not appear any different than data stored in system memory.

Cota, et al. [20] presents a related approach that allows a memory to be used by a single accelerator or within the system cache. Although this approach allows memory to be used both as system memory and as accelerator memory, it comes with limitations. First, the memory can only be mapped to one accelerator at design time, and cannot be reassigned to other accelerators as is possible with the AS. Second, the memory can only be used by the accelerator or by the system at a time—it cannot be used to expose AS data over the system bus.

Unifying AS and system memory presents a more complex challenge, requiring that accelerators and GP-CPUs do not interfere with each other. Current GP-CPU instruction sets (ISAs) may make this impossible: there is not clear way to perform handle operations with system memory using current ISA semantics. Therefore, unifying system and AS memory will probably require a combination of architectural and ISA modifications.

5.3 Dynamic handle allocation

Current uses of the accelerator store have statically assigned handles to accelerators. However, in systems with changing workloads and different accelerators, the handle table configuration will need to change at runtime. A software dynamic handle allocator, similar to software dynamic memory allocators used in virtually all programming languages, would be the best approach for modifying the handle table configuration at runtime. This would allow handles to be automatically unmapped and remapped without exposing the software developers to complexity of runtime allocation.

5.4 ShrinkFit dynamic reprogramming

As previously noted, ShrinkFit is well suited for use in combination with FPGA dynamic reprogramming techniques. Using dynamic reprogramming, accelerators could be dynamically shrunk or expanded as application needs change. Dynamic reprogramming is a challenging problem, and works differently across different FPGA manufacturers and families. For example, entire regions of the FPGA may need to be reprogrammed at once, and multiple accelerators may occupy the same region. Therefore, properly allocating accelerator combinations to each region at runtime is a complicated problem which has limited the use of dynamic reprogramming in the past. However, as dynamic reprogramming techniques improve, the combination of dynamic reprogramming and ShrinkFit could broaden the appeal of FPGA hardware acceleration in mainstream computing.

Appendix A

Helicopter brain prototype

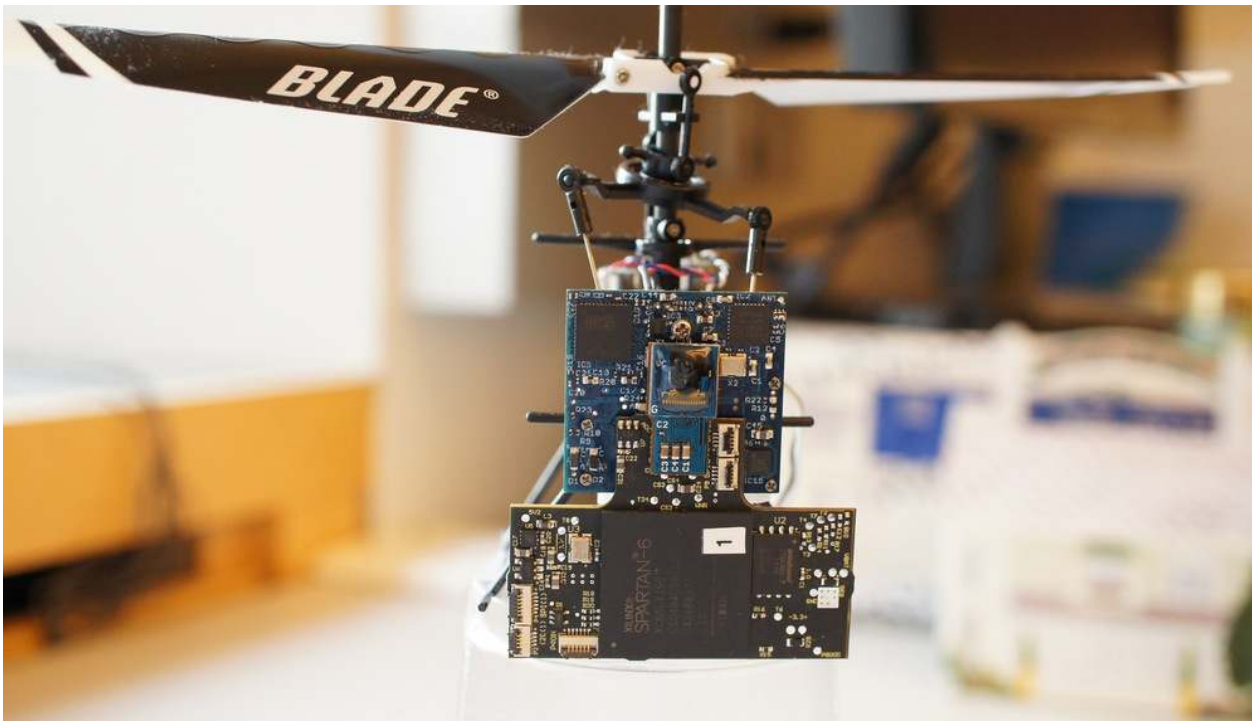


Figure A.1: Helicopter brain prototype attached to helicopter and optical flow camera

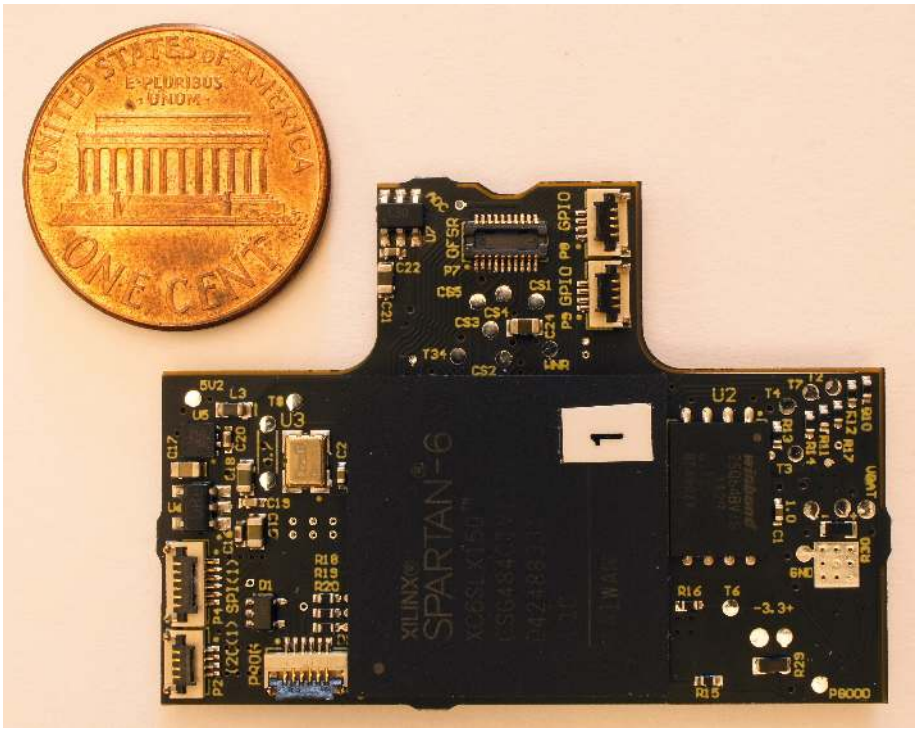


Figure A.2: Helicopter brain prototype front

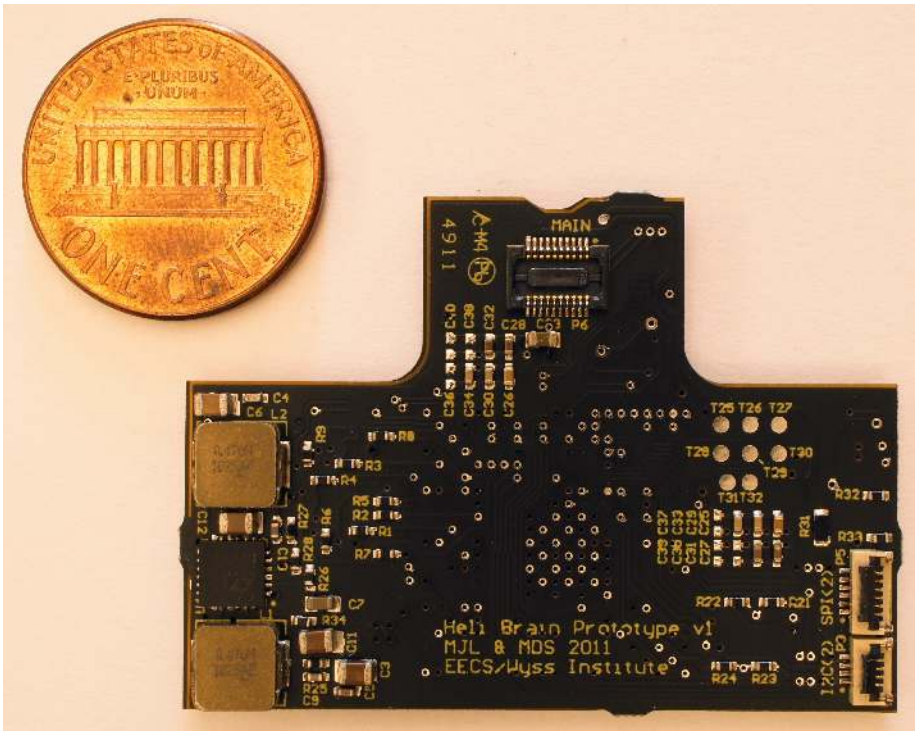


Figure A.3: Helicopter brain prototype back

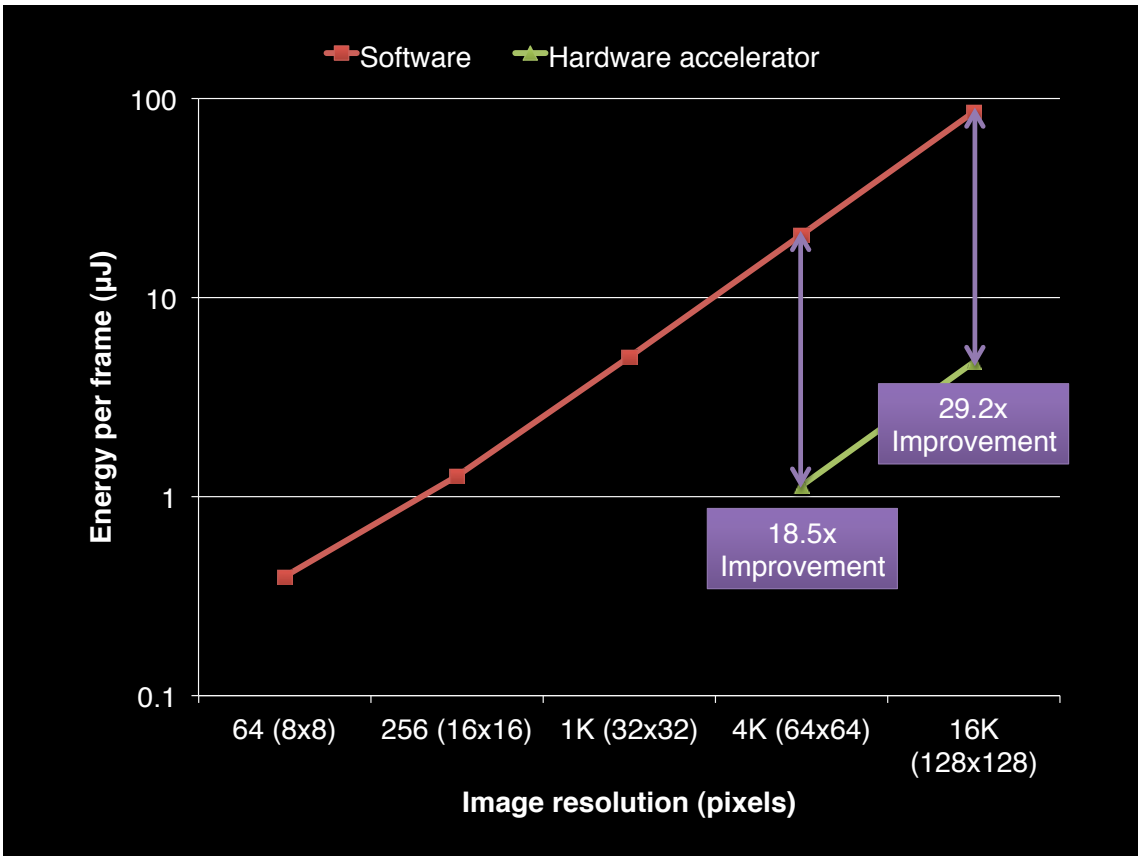


Figure A.4: Optical flow software and hardware accelerator energy consumption

A.1 Brain

Energy is critical due to battery size limits imposed by the helicopter's limited lift. The brain must perform all computation with minimal energy consumption. Computation primarily consists of regularly occurring and computationally intense operations such as optical flow (visual change over time). To maximize energy efficiency, these tasks are implemented as hardware accelerators. Compared to software running on a general purpose processor, 18.5-29.2x energy efficiency improvements are achieved by using hardware accelerators for optical flow (Figure A.4).

The RoboBee brain also includes a general purpose ARM Cortex-M0 microcontroller for infrequent or simple calculations. The Cortex-M0 is extremely low power when active (roughly 85 μW/MHz in ASIC). Although the Cortex-M0 performance is limited, any computations requiring more performance should be realized as accelerators rather than as general purpose core

Figure A.5: MCX2 Toy Helicopter

software.

A.2 Helicopters

The RoboBees team is creating flying bee-sized robots for pollination, search and rescue, and other tasks. The project consists of the body, brain, and colony groups working in parallel. RoboBee body design is in progress, so researchers in the colony and brain teams use toy helicopters (Figure A.5) as a RoboBee approximation. Researchers use the helicopters to try new sensors and develop collaborative strategies. Although the helicopters are larger than the final bee size, they mimic the instability of small vehicle flight and can work together in similar situations.

The helicopters are currently used in two configurations. The first is a stock configuration where small white dots are placed on the helicopter for motion capture. Vicon cameras in the room

use the white dots to locate the helicopters in 3-D space. A stationary computer uses this location data to control each helicopter's motor and steering by radio control.

The second configuration removes the circuit board controlling the helicopter motor and steering and replaces it with two circuit boards and an optical flow sensor ring. The "mainboard" controls the helicopter motors and steering, and includes an AVR32 processor for computation. It also connects to the "optical flow (OF) daughterboard," which measures the amount of visual change seen over time. For example, if two RoboBees were traveling over the same route, but one was flying at twice the speed, it would also see twice the optical flow. The OF daughterboard mainly consists of two AVR32 processors, each with connection pins to an optical flow sensor or sensor ring. Currently, only one of the sensor connections is in use and only one AVR32 processor is processing optical flow calculations. Optical flow sensor rings consist of eight low-resolution, high-speed cameras (optical flow sensors) surrounding the helicopter like a halo.

The HBP merges brain hardware development with software development on the helicopter. The HBP implements an upgradeable copy of the RoboBee brain SoC, and will allow researchers to develop on the RoboBee platform while using helicopters. The HBP will be an in-place replacement for the optical flow daughterboard and communicate with the mainboard and optical flow ring. The HBP will subsume computations from the mainboard as development progresses.

A.3 Objectives

The HBP targets the following objectives:

1. *Brain-Colony collaboration:* The HBP provides a bridge between the brain and colony groups. The brain team developed an early version of the Brain SoC containing the Cortex-M0 GP-CPU and an optical flow accelerator. Meanwhile, the colony group and other researchers in the brain group use helicopters, using GP-CPUs with performance beyond RoboBee GP-CPU capabilities. The HBP allows helicopter teams to design software for the RoboBee brain and use it on helicopters.

2. *Brain SDK:* The HBP helps the brain team evaluate the RoboBee brain as a software development platform. Monitoring the experience of software developers early in the RoboBees

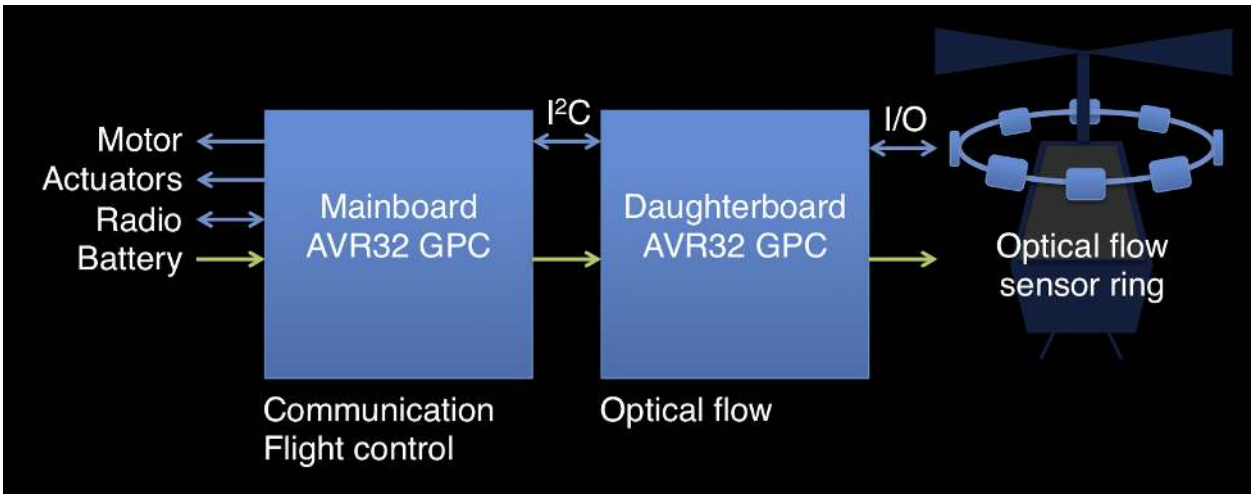


Figure A.6: Original CentEye system architecture

project can reduce programming challenges before design changes cause major regressions.

3. *Sensor evaluation:* The HBP includes several accessible serial ports for interfacing with sensors. This will allow design teams to try out new sensors in the field, without PCB redesign delays.

4. *Accelerator identification:* The brain team developed tools to measure the energy cost of running software on the Cortex-M0 and comparing energy costs to corresponding accelerators. By using the HBP platform rather than the current helicopter configuration, the brain team can identify software to accelerate and compare energy costs for implementing algorithms in software and hardware.

5. *Demos:* The HBP provides fully autonomous flight and demonstrates collaboration between the brain and colony teams.

A.4 System Architecture

The current CentEye configuration connects the mainboard, daughterboard, and optical flow in series (Figure A.6). The battery is connected to the mainboard, and battery power is relayed through the daughterboard and to the optical flow ring.

The HBP replaces the CentEye daughterboard with an FPGA-based board (Figure A.7). The initial HBP bitstream will emulate the daughterboard's optical flow functionality. Additional

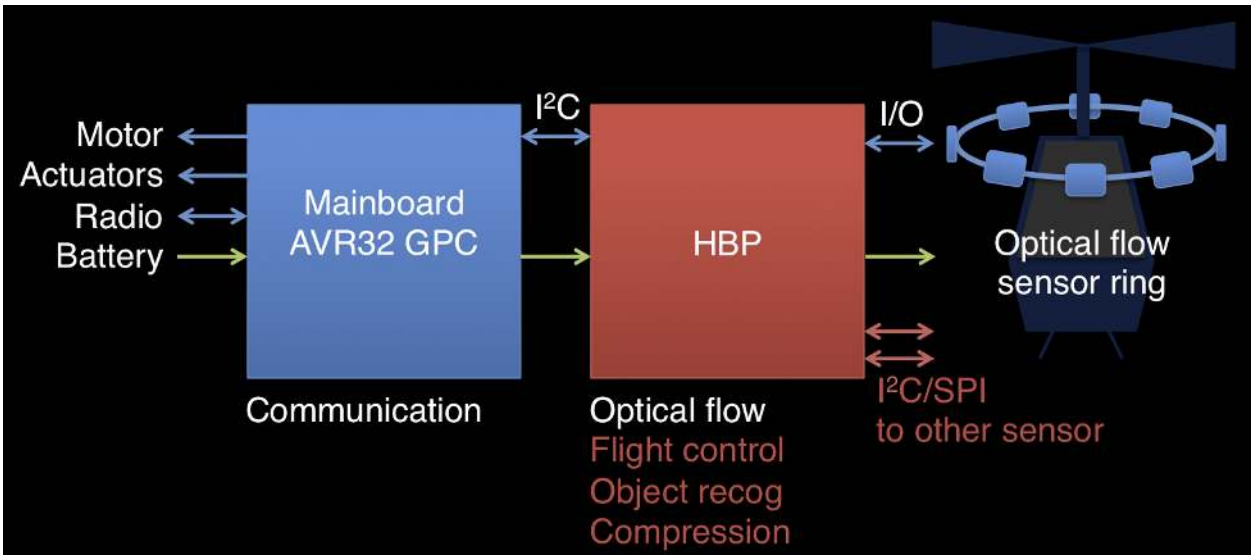


Figure A.7: HBP system architecture
The CentEye mainboard and optical flow ring remain and use the same connectors.

features will be implemented in FPGA logic after.

A.5 HBP Connectors

The HBP maintains the connectors to the mainboard (I2C based) and to the optical flow sensor ring (custom IO protocol). The HBP will also supply two I2C and two SPI connectors for future sensor expansion. In addition, a JTAG connection will be available for FPGA bitstream programming and debugging. Finally, GPIOs will be exposed for any other connectivity needed later.

A.5.1 Connection to mainboard

Figure A.8 shows the connector pinout to the mainboard.

XTRA (1,2): Unconnected

SDA (3): I2C data

SCK (4): I2C clock (driven by mainboard)

SPIxxxx (5-8,11): SPI, unconnected

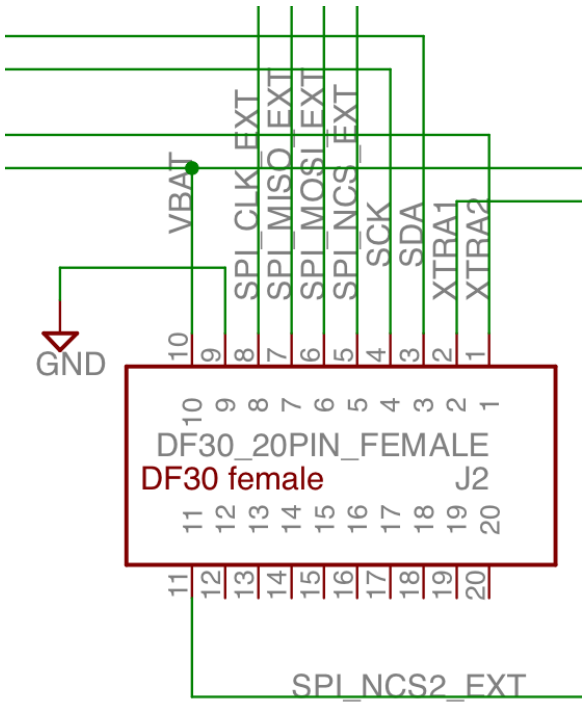


Figure A.8: Interface to mainboard

GND (9)

VBAT (10): unregulated 3.7V-4.0V from helicopter battery

Unused (11-20): Unconnected

A.5.2 Connection to optical flow sensor ring

Figure A.9 shows the connector pinout to the optical flow sensor ring.

AOUT (1): Analog pixel (8-bit grayscale) reading. The HBP's ADC is used to determine the pixel value.

Unused (2): Unconnected

CS7_LED1 (3): Powers an optional LED flash (light) on the sensor ring. Unconnected.

VBAT (4): Unregulated 3.7V-4.0V from helicopter battery.

CS1-5 (5-9): Selects one of the eight sensors on the sensor ring. Three CS signals are always

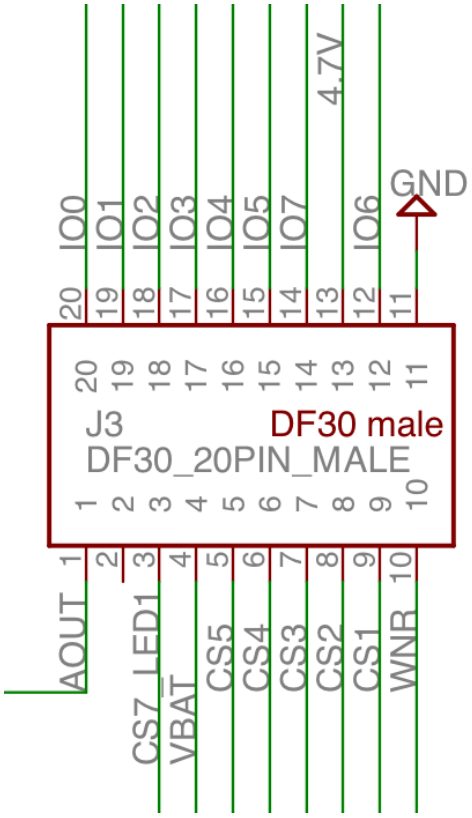


Figure A.9: Interface to optical flow sensor ring

high. Although this signal is 4.7V, the 3.3V provided by the FPGA is sufficient (validated by CentEye).

WNR (10): "Write/NotRead" - used to indicate direction of IO signals.

GND (11)

IO0-7 (12, 14-20): HBP to OFSR 4.7V signals used to send commands to sensors. The HBP can only signal at 3.3V, which is sufficient for the OFSR to detect as high. Because the OFSR never communicates back on these lines, no level converter is needed.

4.7V (13): Regulated 4.7V necessary to power optical flow sensors. This must be regulated on the HBP board.

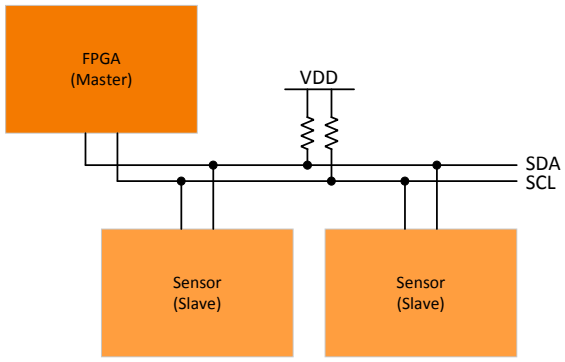


Figure A.10: I2C Interface

A.5.3 JTAG

A Xilinx JTAG connection will be provided for the reconfiguring the FPGA's bitstream (in flash and directly on the FPGA) as well as for debugging. The HBP will not be using the standard 14-pin interface to reduce weight and PCB area. An alternate 6-pin cable will be needed to carry TMS, TCK, TDI, TDO, as well as 3.3V and GND. All four Txx signals are 3.3V.

A.5.4 I2C

The board will offer two I2C serial ports for sensor connectivity. This includes SCL (clock), which is produced by the FPGA. This also includes SDA, which is a shared bidirectional signal. SDA and SCL have pullup resistors (Figure A.10), so devices can only pull SDA to ground when given control of the signal. The HBP connector also supplies 3.3V and GND.

I2C allows multiple sensors over a single port, but multiple ports are provided to improve throughput and simplify connectivity.

A.5.5 SPI

The board will offer two SPI ports for high-throughput sensors. SPI consists of four one-way signals (Figure A.11: SCLK clock generated by the FPGA, MOSI (master out, slave in), MISO (master in, slave out), and $\overline{\text{SS}}$ (slave select). Although SPI can support multiple slaves per master, each slave needs a separate $\overline{\text{SS}}$ line. To reduce design complexity and maximize throughput, two SPI ports are provisioned. 3.3V and GND are also supplied (so a 6-pin connector is necessary).

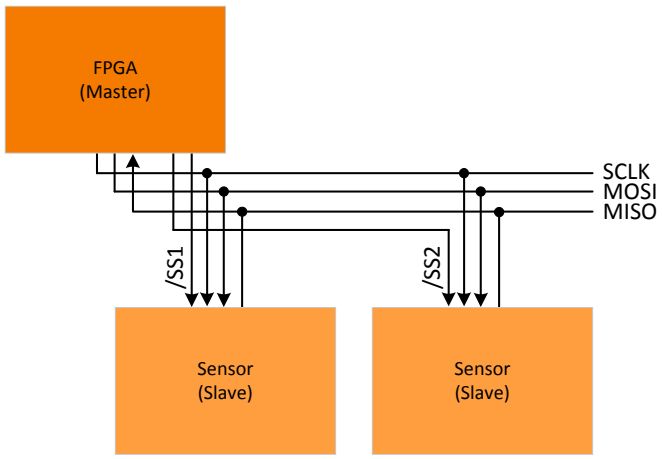


Figure A.11: SPI Interface

A.5.6 GPIO

The board offers eight GPIOs (IO_xxx pins on the FPGA) for input & output. For example, one of the I2C sensors (Gyro + Accelerometer) also has two interrupt pins. These GPIOs are useful for interfacing with any non-I2C/-SPI sensors that may come about (commercially available or through academic projects).

A.6 Components

The HBP will be an in-place replacement of the helicopter's daughterboard. The daughterboard currently implements an optical flow algorithm on an AVR32, so the HBP must also implement the optical flow algorithm. In addition, the HBP must communicate with the mainboard and optical flow sensor ring over separate I2C serial ports. The mainboard connection also provides an unregulated 3.7V battery line.

By stripping the helicopter's body (it's purely cosmetic), replacing the toy's PCB with the CentEye mainboard, and including the optical flow sensor ring, 4.5-6.5g is left for the HBP. The helicopter can fly comfortably up to 4.5g, and struggles up to 6.5g, and will not fly with additional weight. This HBP weight budget includes components (FPGA, flash memory, etc.), solder, and the PCB. Fully assembled, the HBP weighs 4.16g, well below the preferred weight of 4.5g.

Figure A.12 shows how these components connect up in the HBP.

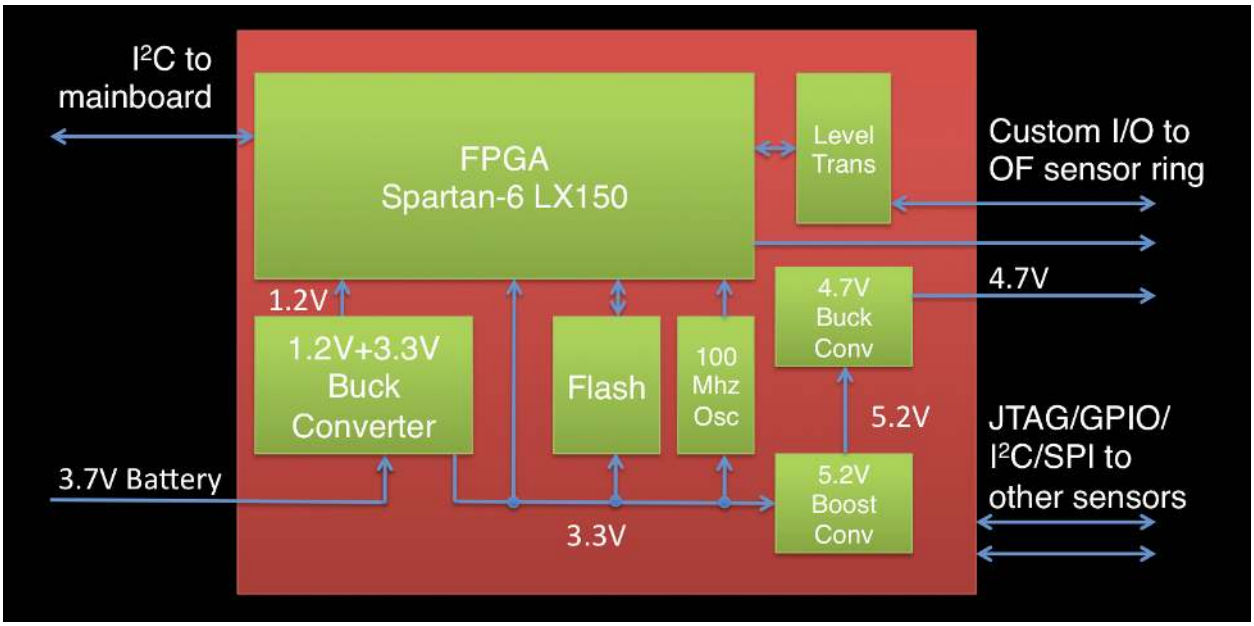


Figure A.12: HBP Architecture

Table A.1: XC6-SLX150 resource utilization for RoboBee brain

System consists of Cortex-M0 GP-CPU and optical flow accelerator

Resource	Utilization %
LUTs (logic)	6%
Registers	1%
DSPs (multiply-accumulate)	17%

A.6.1 FPGA

The FPGA implements the brain's circuitry and can be upgraded in minutes for future versions of the RoboBee brain.

The Xilinx XC6-SLX150 FPGA was selected for its high resource count (Table A.1), and its low energy consumption (Figure A.13 and Figure A.14) and weight. This FPGA gives the brain team room to add new accelerators and provides acceptable flight times. The CSG484 package is the lightest version of the FPGA, weighs 1.3302 g, and measures 19 x 19 mm. The modest IO pin needs are sufficiently handled by the CSG484 package.

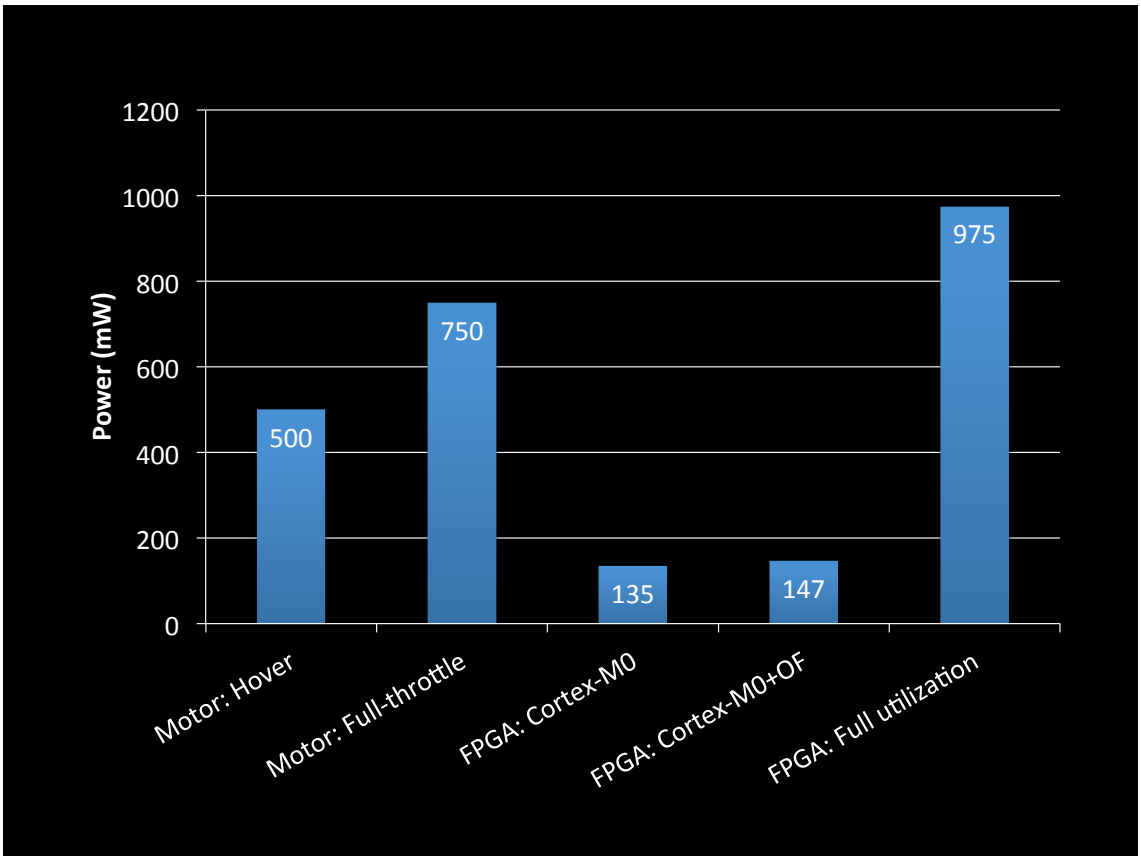


Figure A.13: XC6-SLX150 FPGA power consumption
FPGA power consumption is largely a factor of the resources used. Configurations with just the Cortex-M0 GP-CPU, Cortex-M0 and optical flow accelerator, and full FPGA utilization are shown. The motor currently consumes the most helicopter power and is shown for comparison.

A.6.2 Flash memory

Flash memory is used to configure the FPGA's volatile memory on system startup. The XC6-SLX150 requires slightly more than 4MB to store the configuration, so the minimum flash size is 8MB. The Winbond W25Q64-WSON-ZE was chosen, which contains 8MB of memory and runs entirely on 3.3 V (other flash parts required a higher voltage for writes). The flash weighs 0.11 g and measures 8 x 6 mm.

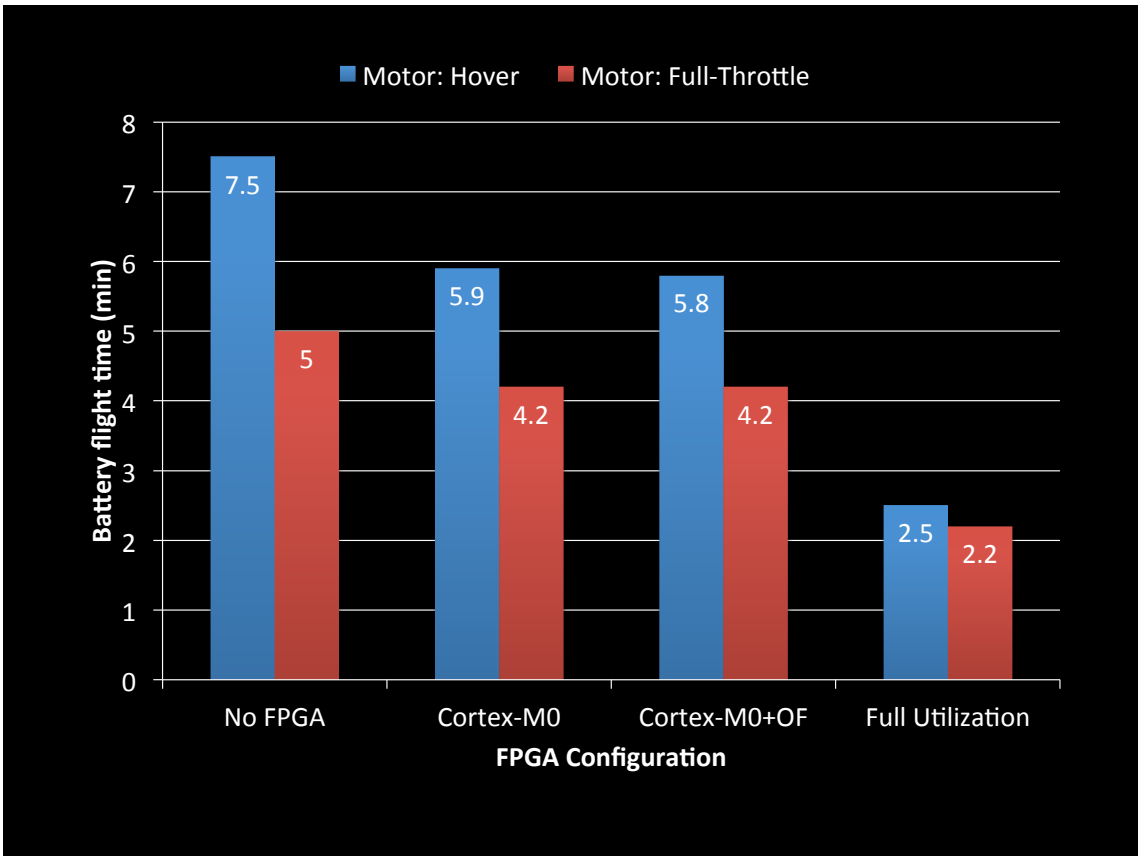


Figure A.14: Helicopter battery life with XC6-SLX150
Estimated battery life for different FPGA configurations is shown. Lifetimes are expected to be between hover and full-throttle times, depending on final weight of HBP and RoboBee task.

A.6.3 1.0V+3.3V Buck Converter

The FPGA requires 3.3 V (provided by the mainboard) for IO and 1.0 V for internal voltage. A buck converter is used to step down from the 3.7V battery to 3.3 V and 1.0 V. The LTC3615 dual voltage regulator was selected for this purpose. This particular model was chosen because it features a devkit with published schematics supporting 1.0V and 3.3V outputs. Also, the Atlys Spartan6 devkit uses similar dual voltage regulators to power the FPGA. Different models are used than in the Atlys because the devkit also generates 1.8V and 2.5V. The LTC3615 can support 3A on both outputs, which matches the regulators used in the Spartan6 devkits.

A.6.4 4.7V Boost+Buck Converter

The original optical flow daughterboard provided a regulated 4.7V for the optical flow sensor ring by using a boost converter (MIC2290) and regulating down to 4.7V (LP3985IM5-4.7/NOPB). The same regulator circuit is used by the HBP.

A.6.5 ADC

An ADC is necessary to determine the intensity of the current pixel read out from the OFSR, since this value is provided in analog. The SPI-controlled AD7276BUJZ ADC was selected for its high bandwidth and precision. The ADC is controlled by the HBP over SPI.

A.6.6 100 MHz Oscillator

The FXO-HC33 HCMOS 100MHz oscillator was selected, which is the replacement part for the oscillator used in the Atlys Spartan6 devkit. It was chosen primarily because it was already proven for the FPGA. In addition, it is relatively small and only requires one external capacitor.

A.6.7 External IO pins

Connections for the following are required:

Mainboard interface: The daughterboard used a Hirose DF30 20 pin female connector to interface with the mainboard, so it is also used by the HBP.

Optical flow sensor ring interface: The daughterboard used a Hirose DF30 20 pin male connector to interface with the OFSR and is reused by the HBP.

I2C: A four pin Hirose interface.

SPI: A six pin Hirose interface.

JTAG: A six pin Hirose interface

GPIO: Eight GPIO pins exposed over two four-pin Hirose interfaces.

A.6.8 PCB

A lightweight PCB of minimum thickness (four layers) is used. Area (weight) must be minimized to fit within budget.

A.7 Helicopter brain prototype implementation

Line #	Designator	Desc	Package Category	Part Count	Package	Mfg Part #	Supplier	Supplier Part #	Packing Container
1	C1,C2,C22	0.1 uF	SMT	3	0402	GRM155R61A104KA01D	Digikey	490-1318-1-ND	Tape
2	C3,C4,C11,C12	47 uF	SMT	4	0805	C2012X5R0J476M	Digikey	445-5987-1-ND	Tape
3	C15,C17,C18,C21,C23,C24	10 uF	SMT	6	0603	ECJ-1VB1A106M	Digikey	587-2562-1-ND	Tape
4	C5,C6,C16,C20,C27,C28	1 uF, 10V 10% X5R 0402	SMT	6	0402	GRM155R61A105KE15D	Digikey	490-3890-1-ND	Tape
5	C7	1 uF, 10V, 10%, X5R 0603	SMT	1	0603	GRM188R61A105KA61D	Digikey	490-1543-1-ND	Tape
6	C9,C10	10 pF, 50V, 5% NP0 0402	SMT	2	0402	GRM1555C1H100JZ01D	Digikey	490-1278-1-ND	Tape
7	C25,C26	4.7 uF cap 6.3 V, X5R, 20%	SMT	2	0402	GRM155R60J475ME87D	Digikey	490-5408-1-ND	Tape
8	C19,C33,C34	10 nF, 25V, 10%, X5R	SMT	3	0402	04023D103KAT2A	Digikey	478-5267-1-ND	Tape
9	C29,C30	220 uF cap 10V, X5R, 10%	SMT	2	0402	GRM155R61A224KE19D	Digikey	490-3910-1-ND	Tape
10	C31,C32	47 uF cap, 25 V, X7R, 10%	SMT	2	0402	GRM155R71E473KA88D	Digikey	490-3254-1-ND	Tape
11	D1	schottky diode (4x)	SMT	1	2 x 2.1mm	NSQA6V8	Digikey	NSQA6V8AW5T2GOSCT-ND	Tape
12	L1,L2	0.47 uH, 5.0A	SMT	2	5.49 x 5.18 x 2mm	IHLP2020BZERR47M01	Newark	05R7400	Tape
13	L3	10 uH	SMT	1	0603	LQM18FN100M00D	Digikey	490-4025-1-ND	Tape
14	R1,R2,R3,R4,R7,R8	100 Ohm	SMT	6	0402	ERJ-2GEJ101X	Digikey	P100JCT-ND	Tape
15	R5,R6	1.8 kOhm	SMT	2	0402	ERJ-2GEJ182X	Digikey	P1.8KJCT-ND	Tape
16	R9,R11,R15,R16,R17	4.7 kOhm	SMT	5	0402	ERJ-2GEJ472X	Digikey	P4.7KJCT-ND	Tape
17	R10,R12,R13	1 kOhm	SMT	3	0402	ERJ-2GEJ102X	Digikey	P1.0KJCT-ND	Tape
18	R14	270 Ohm	SMT	1	0402	ERJ-2GEJ271X	Digikey	P270JCT-ND	Tape
19	R18,R19,R20	200 Ohm	SMT	3	0402	ERJ-2GEJ201X	Digikey	P200JCT-ND	Tape
20	R21,R22,R23,R24	10 kOhm	SMT	4	0402	ERJ-2GEJ103X	Digikey	P10KJCT-ND	Tape
21	R25	845 kOhm, 1%, 1/16W	SMT	1	0402	ERJ-2RKF8453X	Digikey	P845KLCT-ND	Tape
22	R26	187 kOhm, 1%, 1/16W	SMT	1	0402	ERJ-2RKF1873X	Digikey	P187KLCT-ND	Tape
23	R27	348 kOhm, 1%, 1/16W	SMT	1	0402	CRCW0402348KFKED	Digikey	541-348KLCT-ND	Tape
24	R28	523 kOhm, 1%, 1/10W	SMT	1	0402	ERJ-2RKF5233X	Digikey	P523KLCT-ND	Tape
25	R29,R30,R31	0.01 Ohm, 1/4 W, 1%	SMT	3	0603	ERJ-M03NF10MV	Digikey	P10AXCT-ND	Tape
26	R32	16.9 kOhm, 0.05W, 1%, Thick Film	SMT	1	0402	ERJ-2RKF1692X	Digikey	P16.9KLCT-ND	Tape
27	R33	4.99 kOhm, 0.05W, 1%, Thick Film	SMT	1	0402	ERJ-2RKF4991X	Digikey	P4.99KLCT-ND	Tape
28	R34	24 Ohm, 0.1W, 5%	SMT	1	0402	ERJ-2GEJ240X	Digikey	P24JCT-ND	Tape
29	U1	S6LX150-CSG484-3	BGA	1	19 x 19mm	S6LX150-CSG484-3	Xilinx	S6LX150-CSG484-3	Sealed
30	U2	Winbond 64mbit flash memory	Leadless	1	8 x 6mm	W25Q64BVZEIG	Digikey	W25Q64BVZEIG-ND	Tube
31	U3	FXO-HC33 HCMOS 100 Mhz	Leadless	1	3.3 x 2.6mm	FXO-HC335R-100	Digikey	631-1230-1-ND	Tape
32	U4	LTC3615 Regulator	QFN	1	4 x 4mm	LTC3615EUF-1#PBF	Digikey	LTC3615EUF-1#PBF-ND	Tray
33	U5	MIC2290 5.2V Boost Regulator	Leadless	1	2 x 2mm	MIC2290YML TR	Digikey	576-1076-1-ND	Tape
34	U6	4.7V LDO	SMT	1	SOT: 2.92 x 2.84mm	LP3985IM5-4.7/NOPB	Digikey	LP3985IM5-4.7CT-ND	Tape
35	U7	SPI ADC	SMT	1	TSOT23-6	AD7276BUJZ-500RL7	Digikey	AD7276BUJZ-500RL7CT-ND	Tape
36	P6	Hirose DF30_20PIN_FEMALE	Finepitch	1	10.22 x 4.56mm	DF30FC-20DS-0.4V(82)	Digikey	H3859CT-ND	Tape
37	P7	Hirose DF30_20PIN_MALE	Finepitch	1	10.22 x 4.56mm	DF30FC-20DP-0.4V(82)	Digikey	H3864CT-ND	Tape
38	P2,P3,P8,P9	Molex FFC, 4 Pin	Finepitch	4	9.7x6.9mm	52207-0485	Digikey	HFN504CT-ND	Tape
39	P1,P4,P5	Molex FFC, 6 Pin	Finepitch	3	11.7x6.9mm	052207-0685	Digikey	HFN506CT-ND	Tape
	C35,C36,C37,C38,C39,C40	DO NOT POPULATE		6					

Figure A.15: Helicopter brain prototype bill of materials (BOM)

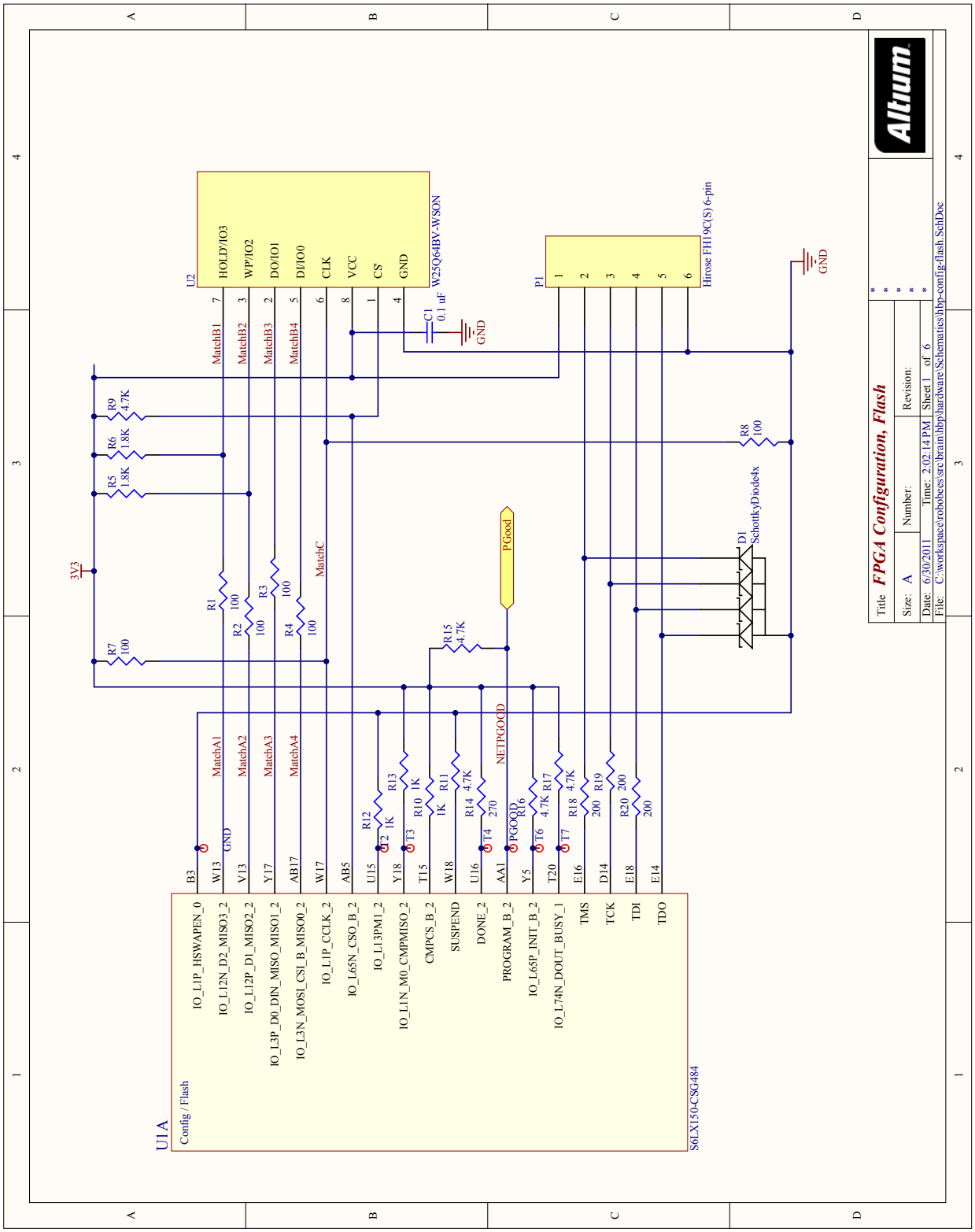


Figure A.16: Helicopter brain prototype schematic: FPGA configuration and flash

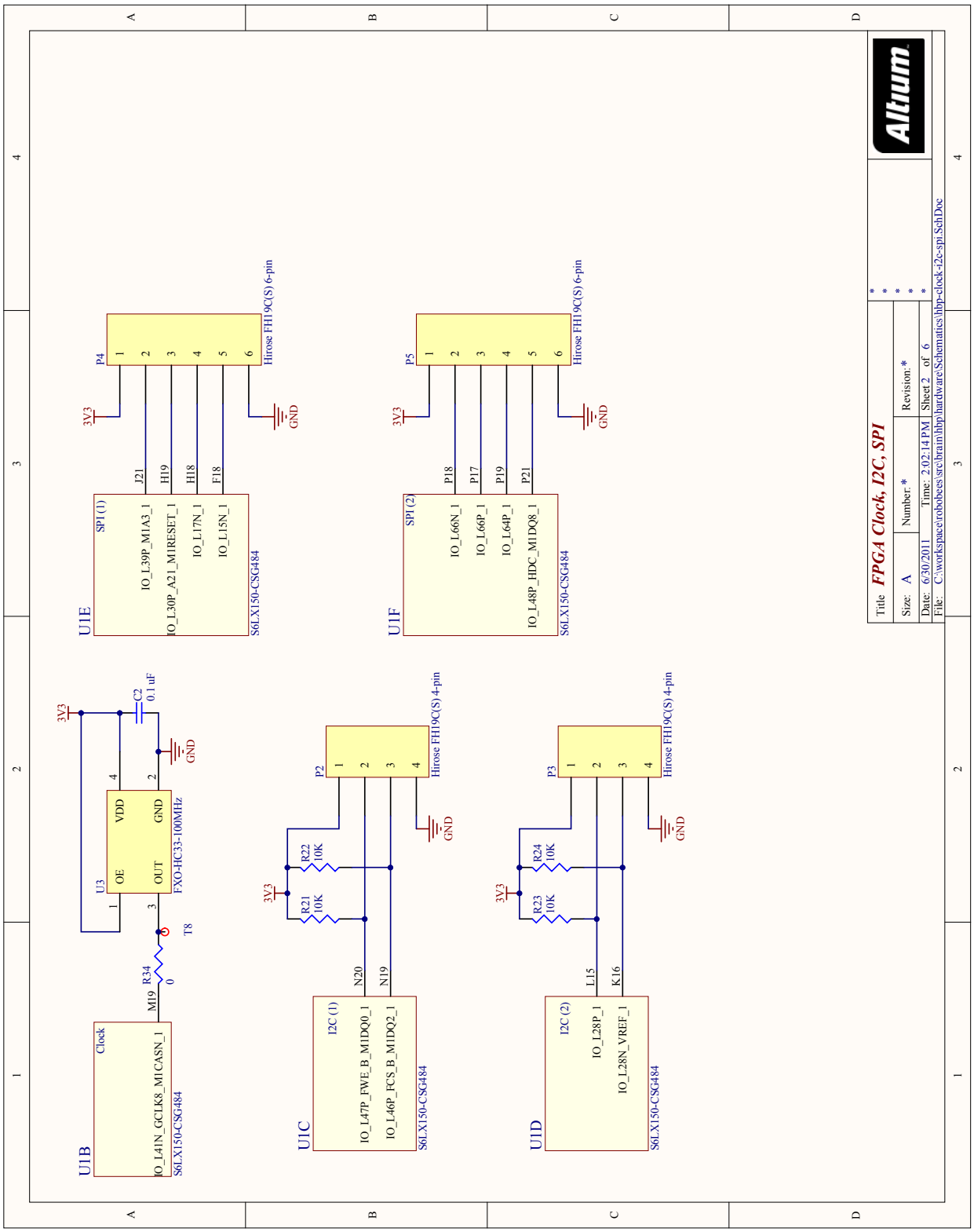


Figure A.17: Helicopter brain prototype schematic: FPGA clock, I2C and SPI interfaces

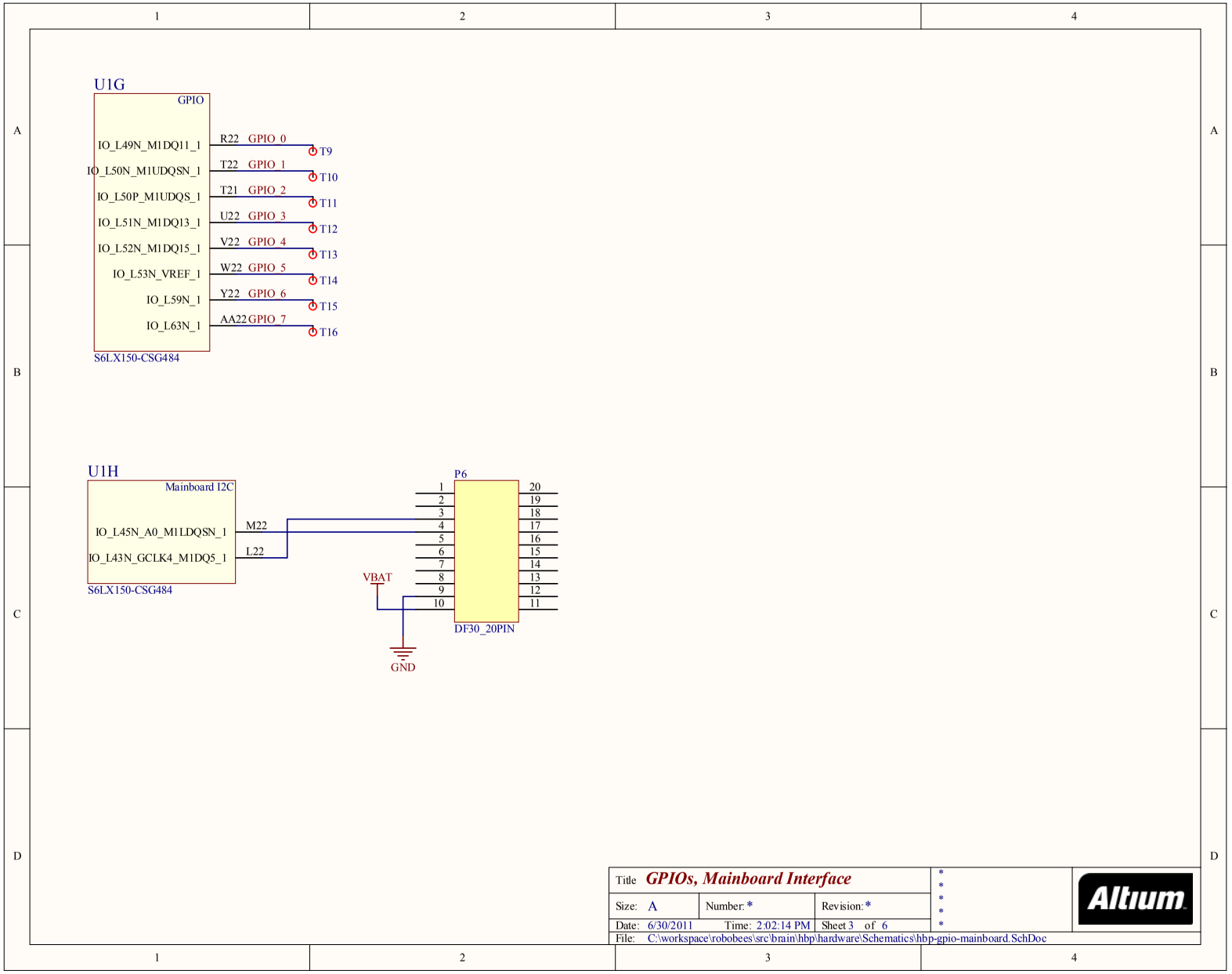


Figure A.18: Helicopter brain prototype schematic: GPIOs and helicopter mainboard interface

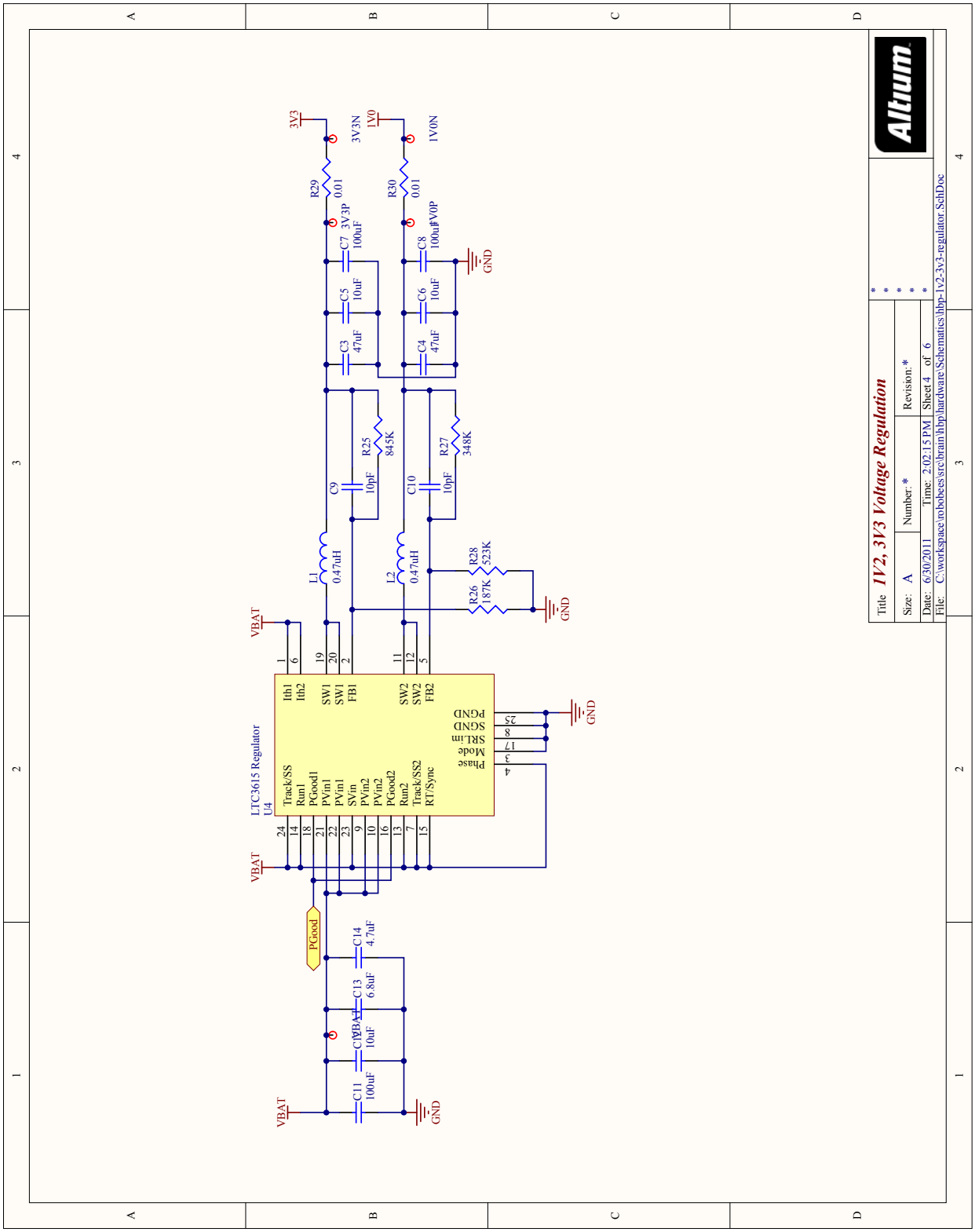


Figure A.19: Helicopter brain prototype schematic: FPGA voltage regulation

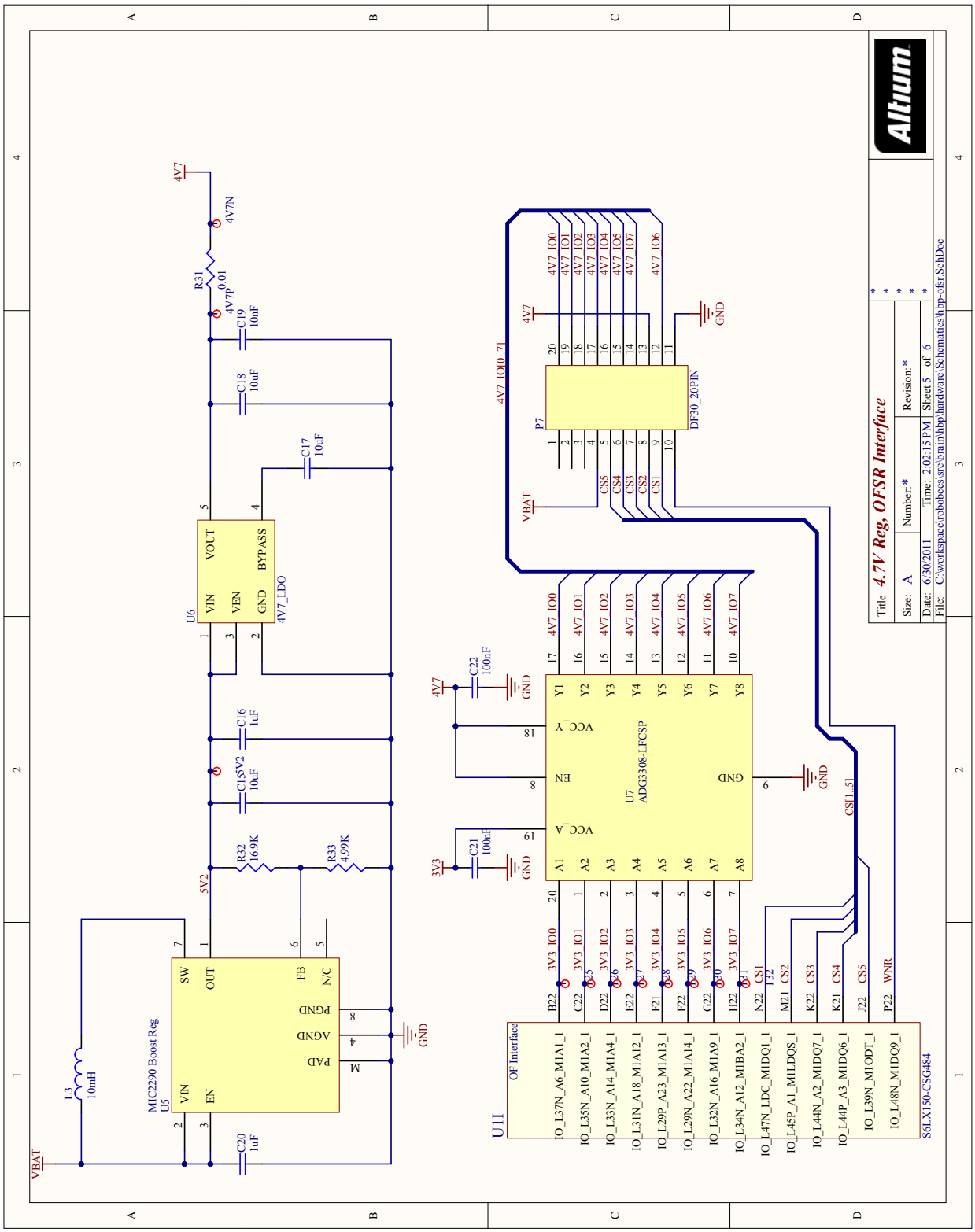


Figure A.20: Helicopter brain prototype schematic: optical flow camera interface

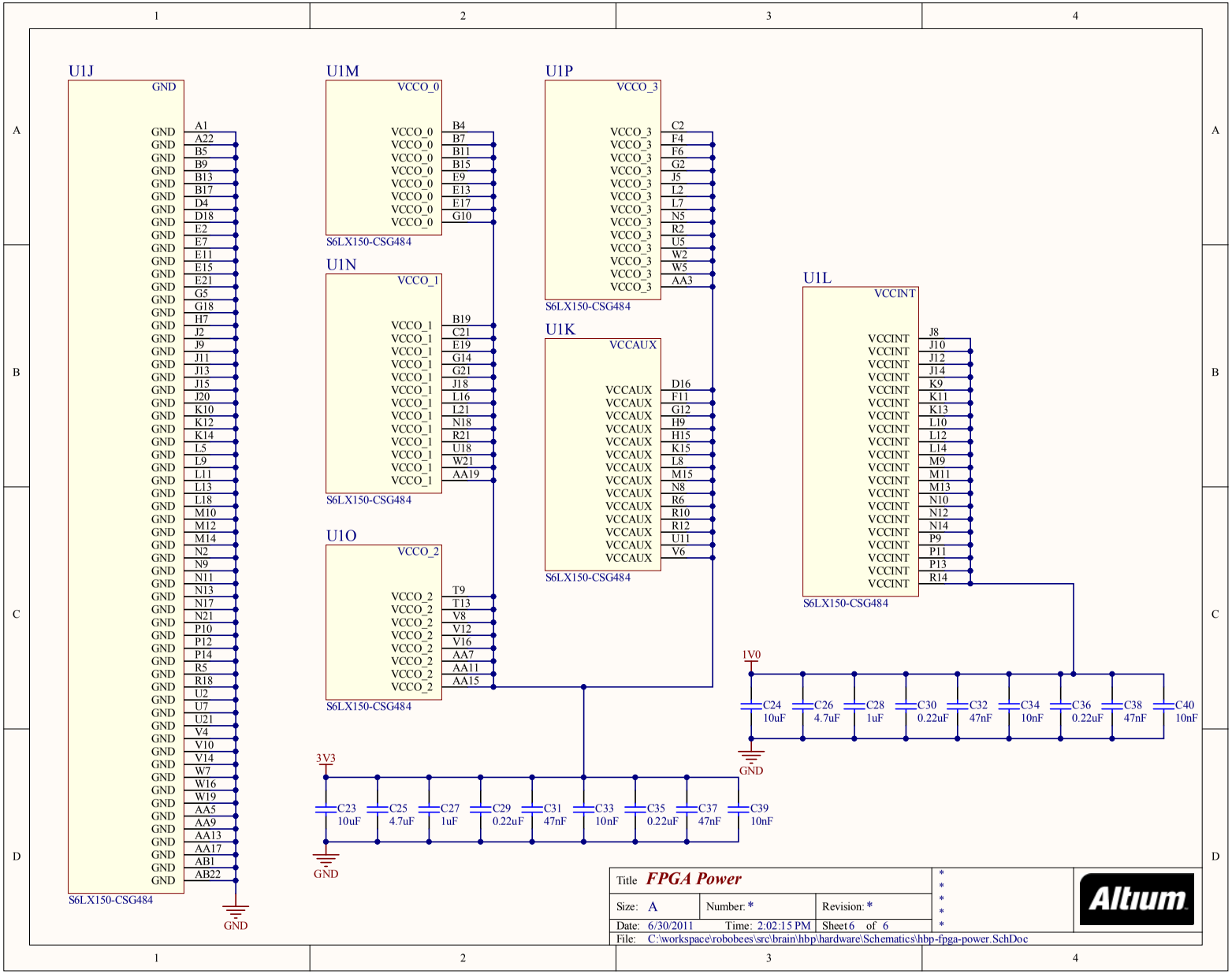


Figure A.21: Helicopter brain prototype schematic: FPGA power connections

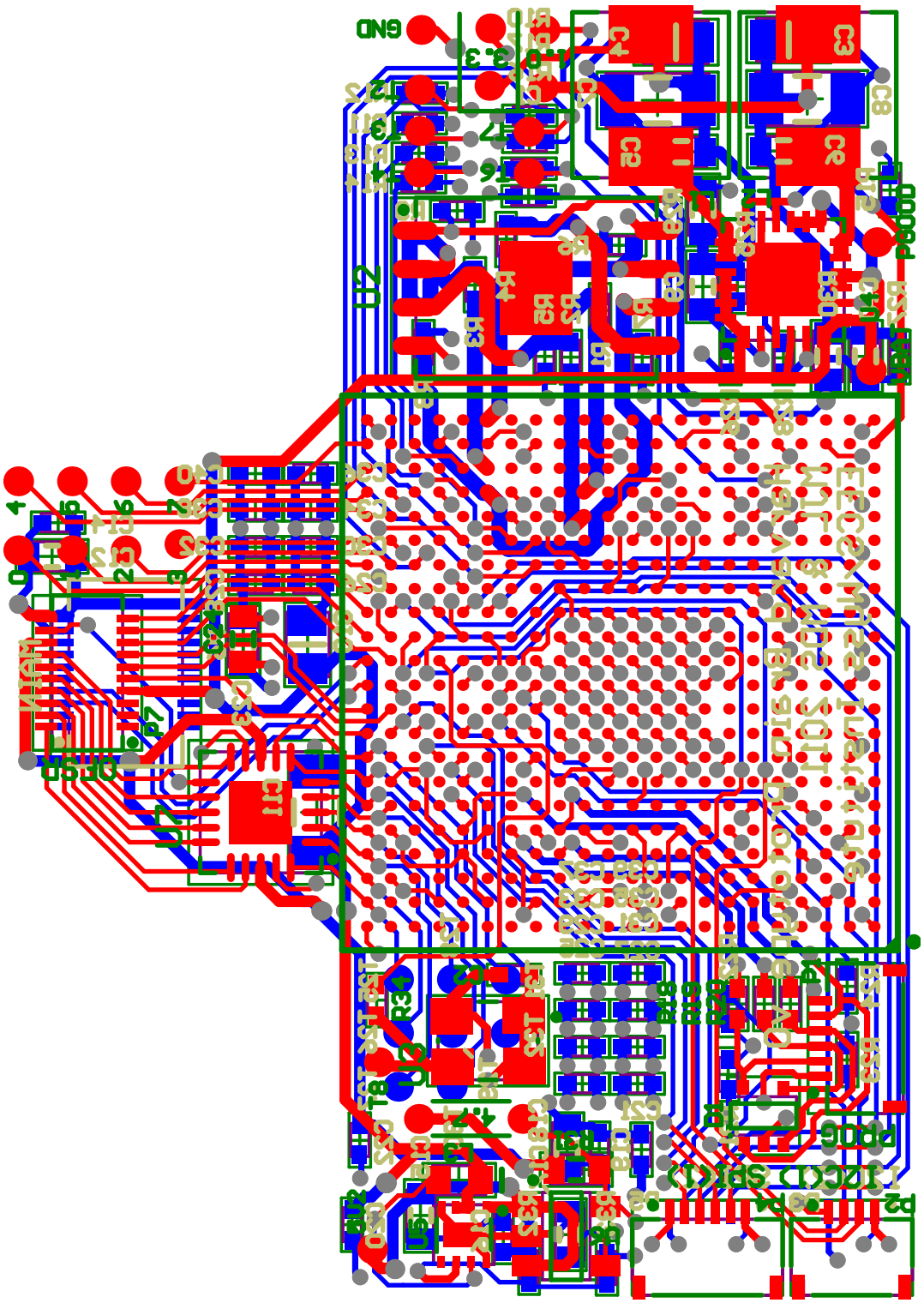


Figure A.22: Helicopter brain prototype layout: all layers

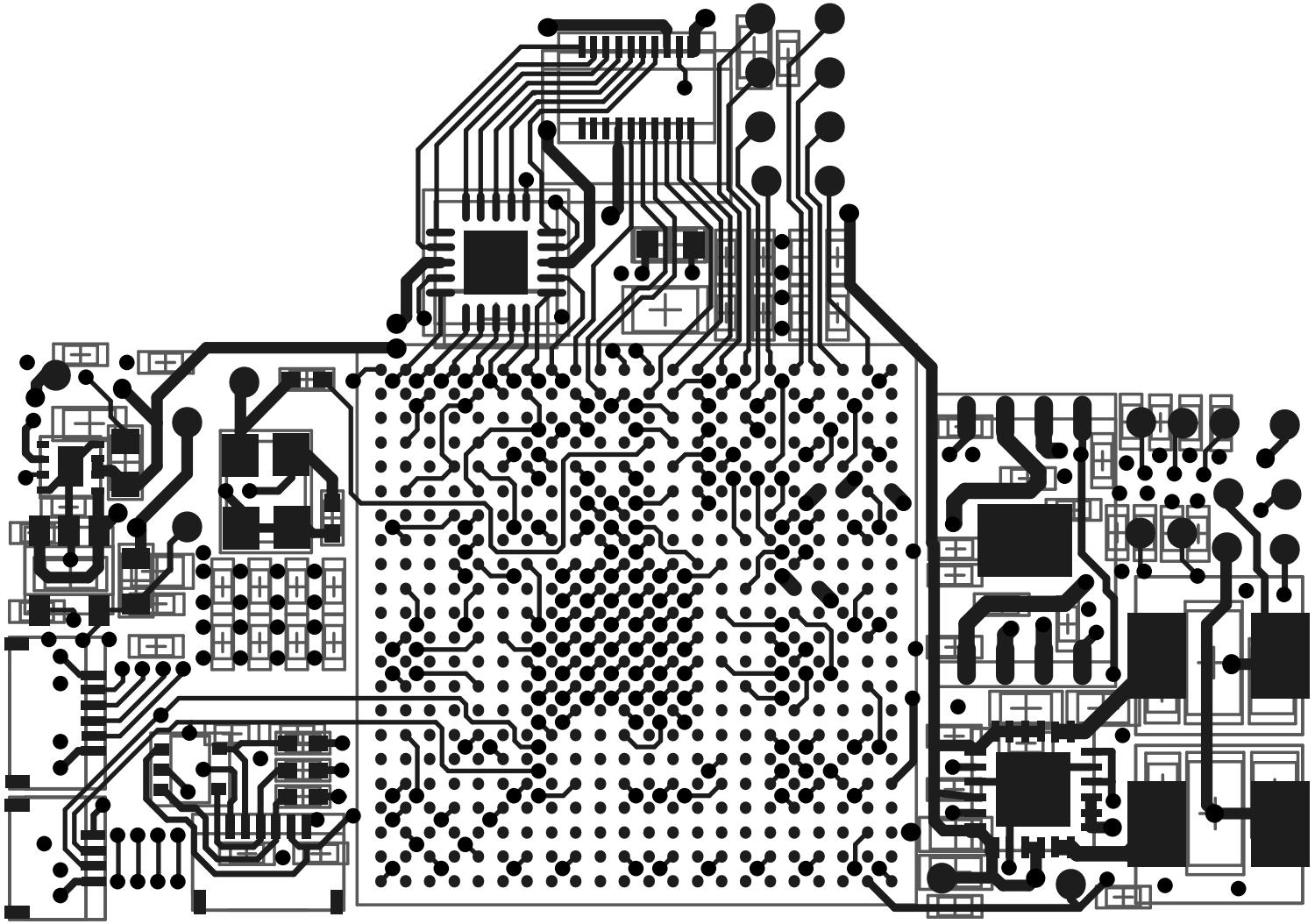


Figure A.23: Helicopter brain prototype layout: front (top) layer

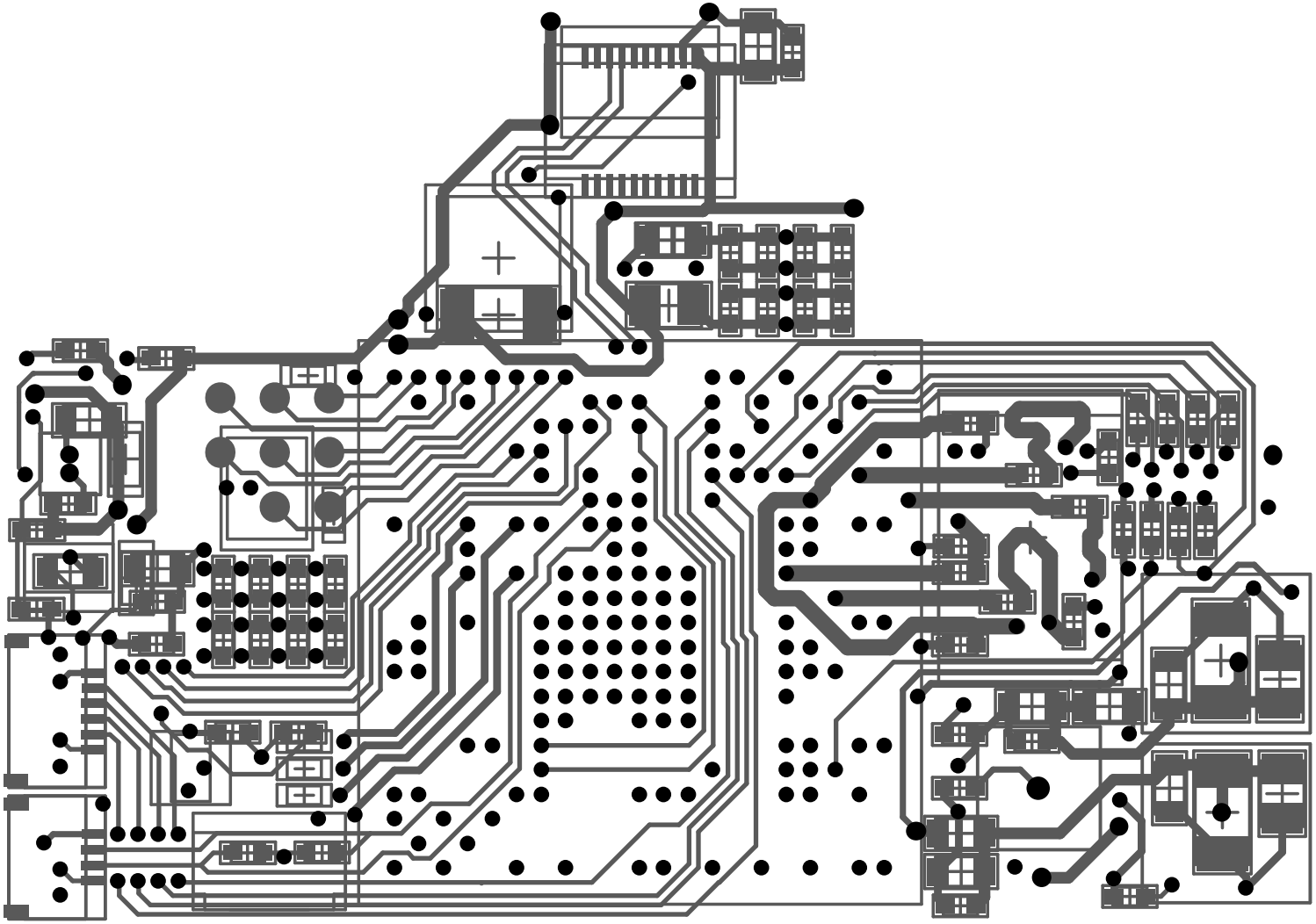


Figure A.24: Helicopter brain prototype layout: back (bottom) layer

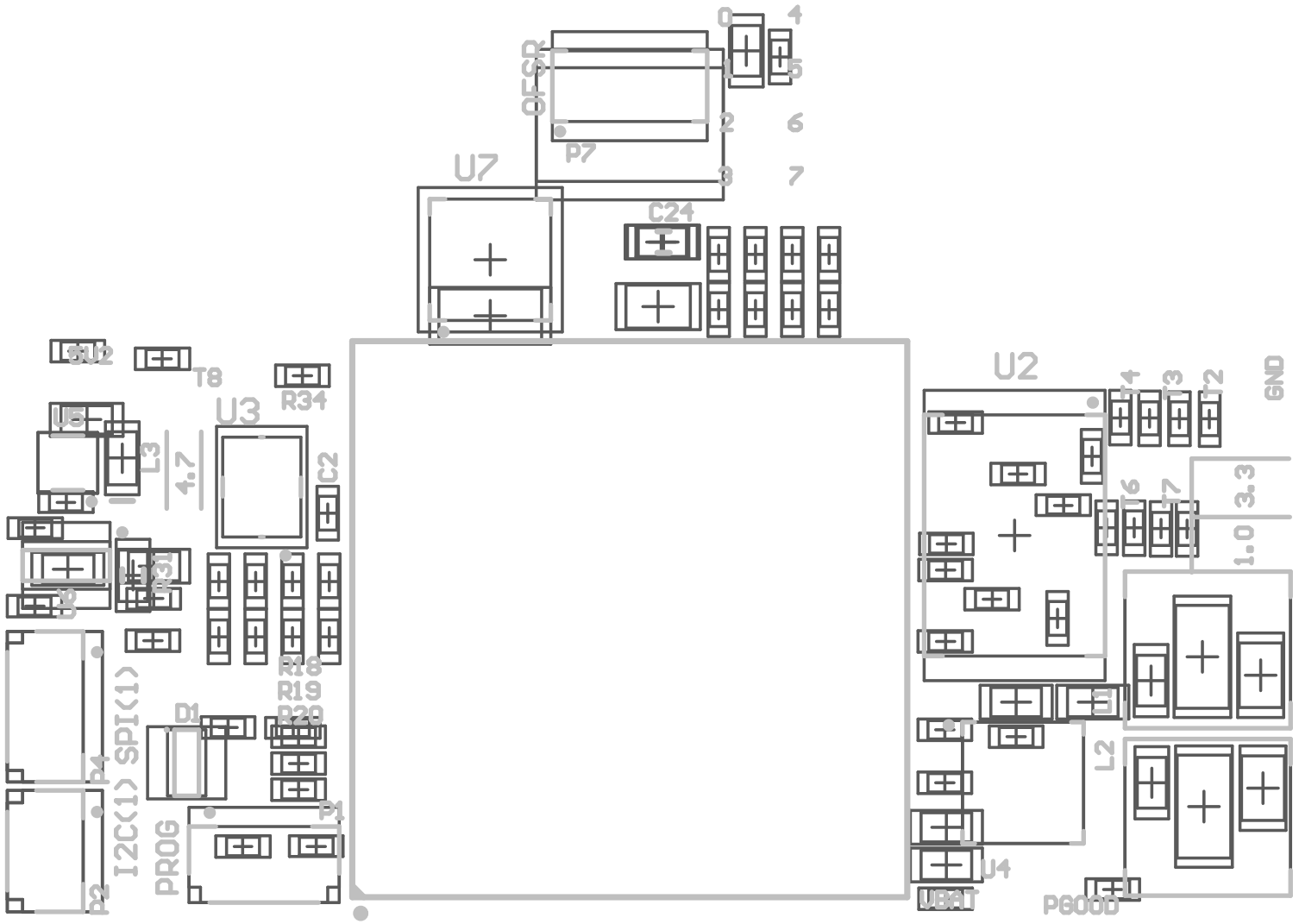


Figure A.25: Helicopter brain prototype components: front

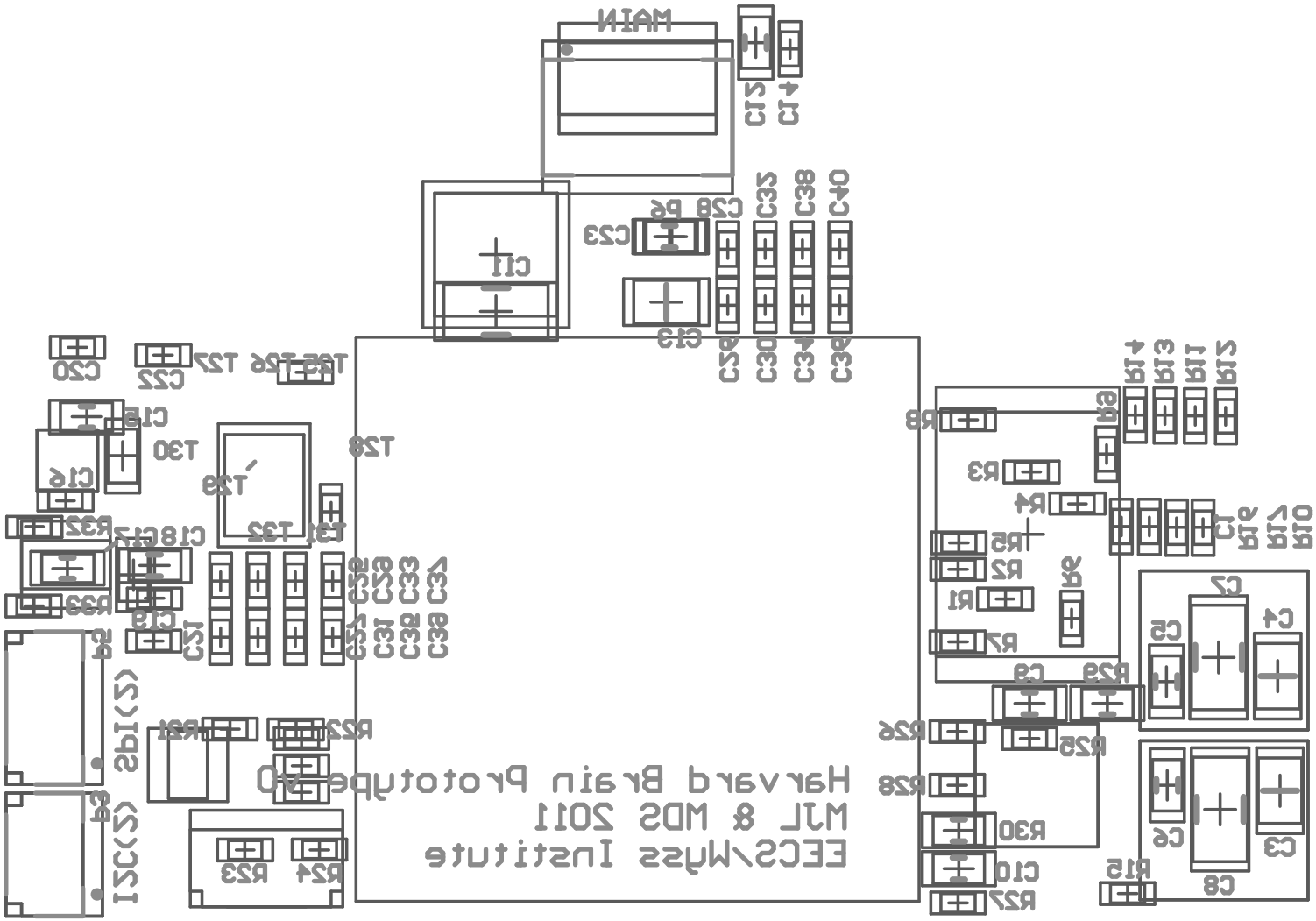


Figure A.26: Helicopter brain prototype components: back

Bibliography

[1] Centeye image sensor. URL `http://centeye.com/projects/robobees`.

[2] Intel Atom processor E6x5C series. URL `http://www.altera.com/devices/processor/intel/e6xx/\proc-e6x5c.html`.

[3] Spartan product tables. URL `http://www.xilinx.com/publications/\matrix/Spartan_Series.pdf`.

[4] Vivado Design Suite. URL `http://xilinx.com/vivado`.

[5] Zynq-7000 all programmable SoC. URL `http://xilinx.com/zynq`.

[6] DSP/BIOS Real-Time Kernel. Technical report, Apr. 2010.

[7] OMAP Technology. Technical report, Apr. 2010.

[8] D. Arora, A. Raghunathan, S. Ravi, M. Sankaradass, N. K. Jha, and S. T. Chakradhar. Software architecture exploration for high-performance security processing on a multiprocessor mobile SoC. In *Design Automation Conference, 2006 43rd ACM/IEEE*, pages 496–501, 2006.

[9] Arvind and others. High-level synthesis: An Essential Ingredient for Designing Complex ASICs. In *ICCAD 2004*.

[10] Benini. Networks on chips: a new SoC paradigm. *Computer*, 35(1):70–78, 2002.

[11] S. Borkar. Design challenges of technology scaling. *IEEE Micro*, 19(4):23–29, 1999.

[12] A. Broder and M. Mitzenmacher. Network applications of bloom filters: A survey. In *Allerton*, 2002.

[13] S. Chang, A. Kirsch, and M. Lyons. Energy and storage reduction in data intensive wireless sensor network applications. Technical Report TR-15-07, Harvard University, 2007.

[14] B. Chen, K.-K. Muniswamy-Reddy, and M. Welsh. Ad-hoc multicast routing on resource-limited sensor nodes. In *REALMAN '06*, pages 87–94. ACM Press, 2006. ISBN 1-59593-360-3. doi: http://doi.acm.org/10.1145/1132983.1132998.

[15] D. Chen, J. Cong, Y. Fan, G. Han, W. Jiang, and Z. Zhang. xPilot: A Platform-Based Behavioral Synthesis System. In *SRC TechCon*, Oct. 2005.

[16] J. Cong, K. Gururaj, and G. Han. Synthesis of reconfigurable high-performance multicore systems. *FPGA*, 2009.

[17] J. Cong et al. High-level synthesis for FPGAs: From prototyping to deployment. *IEEE TCADICS*, 30(4):473–491, 2011.

[18] J. Cong et al. Charm: a composable heterogeneous accelerator-rich microprocessor. ISLPED, 2012.

[19] M. Corporation. `http://www.moteiv.com/products/docs/tmote-sky-datasheet.pdf`, 2007.

[20] E. Cota, P. Mantovani, M. Petracca, M. Casu, and L. Carloni. Accelerator memory reuse in the dark silicon era. *Computer Architecture Letters*, PP(99):1–1, 2013.

[21] M. Dales. Managing a reconfigurable processor in a general purpose workstation environment. *DATE*, 2003.

[22] S. Dharmapurikar and J. Lockwood. Fast and scalable pattern matching for content filtering. In *ANCS '05: Proceedings of the 2005 ACM symposium on Architecture for networking and communications systems*, pages 183–192, New York, NY, USA, 2005. ACM. ISBN 1-59593-082-5. doi: http://doi.acm.org.ezp1.harvard.edu/10.1145/1095890.1095916.

[23] M. Dietzfelbinger, T. Hagerup, J. Katajainen, and M. Penttonen. A reliable randomized algorithm for the closest-pair problem. *J. Algorithms*, 25(1):19–51, 1997. ISSN 0196-6774. doi: http://dx.doi.org/10.1006/jagm.1997.0873.

[24] J. Eyre and J. Bier. The evolution of DSP processors. *Signal Processing Magazine, IEEE*, 17 (2):43–51, Mar. 2000.

[25] W. Fu and K. Compton. An execution environment for reconfigurable computing. *FCCM*, pages 149–158, 2005.

[26] P. Glaskowsky. ATI and Nvidia face off–obliquely.

[27] R. E. Gonzalez. Configurable and Extensible Processors Change System Design. In *Hot Chips 17*. IEEE Computer Society, Aug. 2005.

[28] L. Gwennap. ADAPTEVA: MORE FLOPS, LESS WATTS. Technical report, June 2011.

[29] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz. Understanding sources of inefficiency in general-purpose chips. *ISCA '10: Proceedings of the 37th annual international symposium on Computer architecture*, June 2010.

[30] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz. Understanding Sources of Inefficiency in General-Purpose Chips. In *ISCA 37*, June 2010.

[31] Y. Hasegawa et al. An adaptive cryptographic accelerator for IPsec on dynamically reconfigurable processor. *FPT*, 2006.

[32] P. Hebden and A. R. Pearce. Bloom filters for data aggregation and discovery: a hierarchical clustering approach. In *ICIS '05*, pages 175–180, 2005.

[33] M. Hempstead and others. An Ultra Low Power System Architecture for Sensor Network Applications. In *ISCA '05*, pages 208–219, 2005.

[34] M. Hempstead, N. Tripathi, P. Mauro, G.-Y. Wei, and D. Brooks. An ultra low power system architecture for sensor network applications. In *ISCA '05*, pages 208–219. IEEE Computer Society, 2005. ISBN 0-7695-2270-X. doi: http://dx.doi.org/10.1109/ISCA.2005.12.

[35] M. Hempstead, G.-Y. Wei, and D. Brooks. Navigo: An early-stage model to study power-constrained architectures and specialization. In *ISCA MoBS Workshop*, June 2009.

[36] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. *SIGOPS Oper. Syst. Rev.*, 34(5):93–104, 2000. ISSN 0163-5980. doi: http://doi.acm.org/10.1145/384264.379006.

[37] R. Ho. High Speed and Low Energy Capacitively Driven On-Chip Wires. *Solid-State Circuits, IEEE Journal of*, 43(1):52–60, 2008.

[38] C. Huang and F. Vahid. Transmuting coprocessors: dynamic loading of FPGA coprocessors. In *DAC*, 2009.

[39] S. S. Huang, A. Hormati, D. F. Bacon, and R. Rabbah. Liquid Metal: Object-Oriented Programming Across the Hardware/Software Boundary. In *ECOOP '08: Proceedings of the 22nd European conference on Object-Oriented Programming*, pages 76–103, Berlin, Heidelberg, 2008. Springer-Verlag.

[40] H. Kalte and M. Porrmann. Context saving and restoring for multitasking in reconfigurable systems. *FPL*, 2005.

[41] J. Kelm and S. Lumetta. HybridOS: runtime support for reconfigurable accelerators. *FPGA*, 2008.

[42] J. H. Kelm, D. R. Johnson, M. R. Johnson, N. C. Crago, W. Tuohy, A. Mahesri, S. S. Lumetta, M. I. Frank, and S. J. Patel. Rigel: an architecture and scalable programming interface for a 1000-core accelerator. In *ISCA 2009: Proceedings of the 36th annual international symposium on Computer architecture*, pages 140–151, New York, NY, USA, 2009. ACM.

[43] M. M. Kim, M. Mehrara, M. Oskin, T. Austin, M. M. Kim, M. Mehrara, M. Oskin, and T. Austin. Architectural implications of brick and mortar silicon manufacturing. *ACM SIGARCH Computer Architecture News*, 35(2):244–253, June 2007.

[44] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction. In *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 81, Washington, DC, USA, 2003. IEEE Computer Society.

[45] I. Kuon and J. Rose. Measuring the gap between FPGAs and ASICs. In *FPGA 2006*.

[46] K. Lim, J. Chang, and T. Mudge. Disaggregated memory for expansion and sharing in blade servers. *ACM SIGARCH*, 2009.

[47] Y. Lin, Y. L. Hyunseok, M. Woh, Y. Harel, S. Mahlke, T. Mudge, and C. Chakrabarti. SODA: A Low-power Architecture For Software Radio. In *ISCA-33*, pages 89–101, 2006.

[48] M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng. Merge: a programming model for heterogeneous multi-core systems. In *ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*. ACM Request Permissions, Mar. 2008.

[49] M. Lyons, G. Wei, and D. Brooks. Shrink-fit: A framework for flexible accelerator sizing. *IEEE CAL*, PP(99), 2012.

[50] M. J. Lyons, D. Brooks, and G.-Y. Wei. The accelerator store: A shared memory framework for accelerator-based systems. *ACM TACO*, 8(4):1–22, 2012.

[51] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. Tag: a tiny aggregation service for ad-hoc sensor networks. *SIGOPS Oper. Syst. Rev.*, 36(SI):131–146, 2002. ISSN 0163-5980. doi: http://doi.acm.org/10.1145/844128.844142.

[52] K. Mai, T. Paaske, N. Jayasena, and R. Ho. Smart memories: A modular reconfigurable architecture. *ISCA*, Jan. 2005.

[53] V. Manh Tuan and H. Amano. A preemption algorithm for a multitasking environment on dynamically reconfigurable processor. *ARC*, 2008.

[54] J. Meany. Golomb coding notes. `http://ese.wustl.edu/class/fl06/ese578/GolombCodingNotes.pdf`, 2005.

[55] A. Meixner. Unified microprocessor core storage. *Computing Frontiers*, Jan. 2007.

[56] R. Merritt. ARM CTO: power surge could create 'Dark Silicon'. *EE Times*, 2009.

[57] T. Mudge. Power: A first-class architectural design constraint. *Computer*, 34(4):52–58, 2001. ISSN 0018-9162. doi: http://dx.doi.org/10.1109/2.917539.

[58] M. Muller. 2008 IEEE International Solid-State Circuits Conference - Digest of Technical Papers. In *2008 International Solid-State Circuits Conference - (ISSCC)*, pages 32–37. IEEE, 2008.

[59] J.-K. Peir, S.-C. Lai, S.-L. Lu, J. Stark, and K. Lai. Bloom filtering cache misses for accurate data speculation and prefetching. In *ICS '02*, pages 189–198. ACM Press, 2002. ISBN 1-58113-483-5. doi: http://doi.acm.org/10.1145/514191.514219.

[60] N. Pinto, D. Doukhan, J. J. DiCarlo, and D. D. Cox. A High-Throughput Screening Approach to Discovering Good Forms of Biologically Inspired Visual Representation. *PLoS Comput Biol*, 5(11):e1000579, Nov. 2009.

[61] P. Pratim. Performance evaluation and design trade-offs for network-on-chip interconnect architectures. *Computers, IEEE Transactions on*, 54(8):1025–1040, 2005.

[62] A. Roth. Store vulnerability window (SVW): Re-execution filtering for enhanced load optimization. In *ISCA '05*, pages 458–468. IEEE Computer Society, 2005. ISBN 0-7695-2270-X. doi: http://dx.doi.org/10.1109/ISCA.2005.48.

[63] K. Sayood. *Introduction to Data Compression*. Morgan Kaufmann Publishers, second edition, 2000.

[64] SD Association. SD Speed Class. URL `http://www.sdcard.org/developers/tech/speed_class`.

[65] M. Shalan and V. I. Mooney. Hardware support for real-time embedded multiprocessor system-on-a-chip memory management. In *Hardware/Software Codesign, 2002. CODES 2002. Proceedings of the Tenth International Symposium on*, pages 79–84, 2002.

[66] Y. Shan et al. FPMR: MapReduce framework on FPGA. In *FPGA*, 2010.

[67] D. E. Shaw, R. O. Dror, J. K. Salmon, J. P. Grossman, K. M. Mackenzie, J. A. Bank, C. Young, M. M. Deneroff, B. Batson, K. J. Bowers, E. Chow, M. P. Eastwood, D. J. Ierardi, J. L. Klepeis, J. S. Kuskin, R. H. Larson, K. Lindorff-Larsen, P. Maragakis, M. A. Moraes, S. Piana, Y. Shan, and B. Towles. Millisecond-scale molecular dynamics simulations on Anton. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–11, New York, NY, USA, 2009. ACM.

[68] S. S. Stone, J. P. Haldar, S. C. Tsao, W. m. W. Hwu, B. P. Sutton, and Z. P. Liang. Accelerating advanced MRI reconstructions on GPUs. *J. Parallel Distrib. Comput.*, 68(10):1307–1318, 2008.

[69] D. Talla. Evaluating VLIW and SIMD Architectures for DSP and Multimedia Applications. Technical report.

[70] C. Taylor, A. Rahimi, J. Bachrach, H. Shrobe, and A. Grue. Simultaneous localization, calibration, and tracking in an ad hoc sensor network. In *IPSN '06*, pages 27–33. ACM Press, 2006. ISBN 1-59593-334-4. doi: http://doi.acm.org/10.1145/1127777.1127785.

[71] M. Ullmann et al. An FPGA run-time system for dynamical on-demand reconfiguration. *IPDPS*, 2004.

[72] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor. Conservation cores: reducing the energy of mature computations. In *ASPLOS '10: Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, pages 205–218, New York, NY, USA, 2010. ACM.

[73] G. Venkatesh et al. Conservation cores: reducing the energy of mature computations. *ASPLOS*, 2010.

[74] A. Wang, B. H. Calhoun, and A. P. Chandrakasan. *Sub-threshold Design for Ultra Low-Power Systems*. Springer, 2006.

[75] J. R. Wernsing and G. Stitt. Elastic computing: a framework for transparent, portable, and adaptive multi-core heterogeneous computing. In *LCTES*, 2010.

[76] M. Woh, Y. Lin, S. Seo, S. Mahlke, T. Mudge, C. Chakrabarti, R. Bruce, D. Kershaw, A. Reid, M. Wilder, and K. Flautner. From SODA to scotch: The evolution of a wireless baseband processor. In *MICRO '08: Proceedings of the 2008 41st IEEE/ACM International Symposium on Microarchitecture*, pages 152–163, Washington, DC, USA, 2008. IEEE Computer Society.