A Case Study of Multi-Threading in the Embedded Space

Greg Hoover
University of California, Santa Barbara
Engineering I
Santa Barbara, California 93106
ghoover@ece.ucsb.edu

Forrest Brewer
University of California, Santa Barbara
Engineering I
Santa Barbara, California 93106
forrest@ece.ucsb.edu

Timothy Sherwood
University of California, Santa Barbara
Engineering I
Santa Barbara, California 93106
sherwood@cs.ucsb.edu

ABSTRACT

The continuing miniaturization of technology coupled with wireless networks has made it feasible to physically embed sensor network systems into the environment. Sensor net processors are tasked with the job of handling a disparate set of interrupt driven activity, from networks to timers to the sensors themselves. In this paper, we demonstrate the advantages of a tiny multi-threaded microcontroller design which targets embedded applications that need to respond to events at high speed. While multi-threading is typically used to improve resource utilization, in the embedded space it can provide zero-cycle context switching and interrupt service threads (IST), enabling complex programmable control in latency constrained environments. To explore the advantages of multi-threading on these embedded problems, we have implemented in hardware a family of controllers supporting eight dynamically interleaved threads and executing the AVR instruction set. This allows us to carefully quantify the effects of threading on interrupt latency, code size, overall processor throughput, cycle time, and design area for complete designs with different numbers of threads.

Categories and Subject Descriptors

B.8.2 [**Hardware**]: Performance and Reliability—*performance analysis and design aids*; B.5 [**Hardware**]: Register-Transfer-Level Implementation

General Terms

Performance, Design

Keywords

multi-threading, embedded architecture

1. INTRODUCTION

The continuing miniaturization of technology enables small systems to be innocuously embedded into our physical environment. Such systems open the possibility of early detection of structural failure, tracking microclimates in forest canopies, and observing migratory patterns of any number of species [16, 11, 29]. In an ideal world, these tiny digital systems would be operating a wireless network, handling sensor readings, controlling electro-statically-activated devices, processing software updates, and performing distributed computations. To handle all of these functions at the required throughput, we argue for the use of dynamic multi-threading at all levels of the microcontroller design.

At first this concept may seem counterintuitive as most multi-threaded architectures were developed to better-exploit an *abundance* of resources [25], something that these systems most certainly do not have. Indeed, because most modern hardware-threaded systems are superscalar designs with support for speculative or even out-of-order execution, a very common belief is that adding hardware support for threading will increase complexity more than is sensible in the microcontroller space. Only by carefully quantifying both the circuit-level overhead of multi-threading and the software-level advantages of its application can we truly understand the tradeoffs involved in embedded multi-threaded systems.

This paper presents the architecture and implementation details of our multi-threaded microcontroller designed with these concerns in mind. Our design, JackKnife, aims to show that threading can be a winning design point in this space, providing a synthesizable, multi-threaded, pipelined microcontroller supporting the AVR instruction set. JackKnife is highly modular and customizable, supporting the inclusion of custom peripherals via an extensible bus architecture. The current implementation supports up to 8 dynamically interleaved threads, each with independent processor contexts. The performance of our implementation in 0.15μm TSMC process synthesizes at over 400MHz. We show at least an order of magnitude increase in performance over currently available AVR designs which, for many instructions, require multiple cycles per instruction and have a maximum clock frequency of 20MHz.

We show that a practical embedded multi-threaded design is possible (even for an ISA as complex as AVR) and through a detailed hardware-software implementation study we demonstrate that:

- zero-cycle context switching enables extremely low latency interrupt response which could then be exploited to decrease the required frequency.
- for a fully synthesizable design (including control and

datapaths), the area overhead to implement multi-threading is minimal – only 20% per added thread.

- not only does the latency response improve, multi-threaded interrupt code actually requires both less total storage (50% on average) and less total instructions executed (50% on average) than unthreaded code

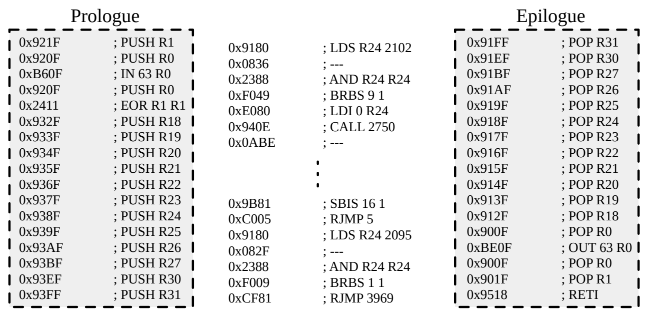
If supported as a first class design constraint at all levels, threading and pipelining the AVR does not add significantly to the complexity or area of the design. The major complexity comes in supporting true zero-cycle interrupts and in the interaction between in-flight interrupts and schedule-effecting instructions. However, by implementing the control using the pyPBS control synthesis tool, the complexity exposed to the designer is kept to a minimum. The remainder of this paper is organized as follows: Section 2 discusses the motivation for and contributions of this work. Section 3 describes the architectural features of the processor core pertaining to interleaving, pipeline structure, scheduling, interrupts and synchronization. The memory architecture is presented in Section 4 with specifics on instruction fetch, the register file, and the extensible bus (I/O). Section 5 outlines implementation details with discussion of multi-threading effects and synthesis results. Prior art in the field of multi-threading is presented in Section 6. Finally, we summarize our findings in Section 7.

2. MOTIVATION

The availability of small electronic sensors and embedded computing platforms has already provided the means for exploring new areas in network sensors. In general, sensor systems tend to be small in size, data-intensive, diverse in design and use, and exhibit limited physical parallelism [8]. These systems must meet size, reliability, and longevity requirements [11], while providing communication and sensor interfaces, data processing and storage, low-latency response times, and power management. Existing network sensor architectures have been built around commercially available microcontroller devices such as the AVR microcontroller [8], the Hitachi SH1[11, 29], and the ARM7 [12].

While traditional embedded systems and sensor nets share many similarities (power and cost constraints, small amounts of storage, etc.), a sensor net processor is called on to be truly general purpose. In particular, sensor net processors juggle many different interrupt driven tasks, from timers to sensors to the network. As interrupts are one of the most common tasks, they need to be handled efficiently. Unfortunately, in a single threaded design, managing the complexity of program execution becomes difficult, and the bookkeeping work of transferring control between processes consumes significant time and space.

Figure 1 highlights the prologue and epilogue code segments for an instruction service routine on single threaded AVR microcontroller. While the size of these segments can vary, the routine presented is not atypical. Such save/return code segments can dominate the execution, resulting in inefficient use of resources. Each added instruction affects, not only, the program footprint, but also execution latency; and the added delay serves to further constrain the rate at which the system can process events. While Figure 1 motivates the need for a more efficient interrupt management method, a quantitative analysis of this problem is presented in Section 5.



```
Prologue                                                          Epilogue
0x921F   ; PUSH R1       0x9180   ; LDS R24 2102                  0x91FF   ; POP R31
0x920F   ; PUSH R0       0x0836   ; ---                           0x91EF   ; POP R30
0xB60F   ; IN 63 R0      0x2388   ; AND R24 R24                   0x91BF   ; POP R27
0x920F   ; PUSH R0       0xF049   ; BRBS 9 1                      0x91AF   ; POP R26
0x2411   ; EOR R1 R1     0xE080   ; LDI 0 R24                     0x919F   ; POP R25
0x932F   ; PUSH R18      0x940E   ; CALL 2750                     0x918F   ; POP R24
0x933F   ; PUSH R19      0x0ABE   ; ---                           0x917F   ; POP R23
0x934F   ; PUSH R20                                               0x916F   ; POP R22
0x935F   ; PUSH R21                 ⋮                             0x915F   ; POP R21
0x936F   ; PUSH R22                                               0x914F   ; POP R20
0x937F   ; PUSH R23      0x9B81   ; SBIS 16 1                     0x913F   ; POP R19
0x938F   ; PUSH R24      0xC005   ; RJMP 5                        0x912F   ; POP R18
0x939F   ; PUSH R25      0x9180   ; LDS R24 2095                  0x900F   ; POP R0
0x93AF   ; PUSH R26      0x082F   ; ---                           0xBE0F   ; OUT 63 R0
0x93BF   ; PUSH R27      0x2388   ; AND R24 R24                   0x900F   ; POP R0
0x93EF   ; PUSH R30      0xF009   ; BRBS 1 1                      0x901F   ; POP R1
0x93FF   ; PUSH R31      0xCF81   ; RJMP 3969                     0x9518   ; RETI
```

Figure 1: Assembly code from an AVR interrupt service routine highlighting the overhead of prologue and epilogue code segments.

In traditional microcontrollers, software developers cannot be shielded from the fact that interrupts are typically both complex to program and slow to respond. While the peripheral sets of modern microcontrollers continue to expand, support for the growing complexity of interrupt processing does not. These devices are typically limited to, at most, one fast-interrupt source. This imposes limitations on embedded software, requiring the overhead of a task-manager or similar OS-level component [8, 16, 15].

The main idea behind this paper is to explore the effectiveness of multi-threading in the embedded space. While the case for interrupt service threads has been made before, no one to date has conducted a careful study of embedded multi-threading that quantified the ramifications from the circuit level all the way to the software layers. A multi-threaded design will only make sense if the software gains are thoroughly weighed against the hardware overheads. To flush out the important tradeoffs we have fully implemented a family of threaded processors which are binary compatible with the Atmel AVR instruction set. We quantify the area and timing ramifications of our design, and more importantly we demonstrate how it scales with the number of hardware threads. However, before we get to our results we need to present the details of our micro-architecture and describe the ways which made multi-threading work in this extremely resource constrained environment.

3. PROCESSOR ARCHITECTURE

JackKnife models its architecture after that of the AVR, providing advanced capabilities while maintaining simplicity and software compatibility. Figure 2 provides a functional overview of the processor architecture, illustrating the division of the six pipeline stages. A memory-mapped infrastructure is used throughout the design, reducing design complexity by providing a uniform interface to system peripherals and memory. By supporting interleaved multi-threading, JackKnife provides implicit sharing of data path resources for increased throughput and low-latency response to dynamic events. Extensions to the original AVR pipeline result in over an order of magnitude increase in clock rate, further increasing system throughput. Additionally, JackKnife includes a custom scheduler, dynamic interrupt handling, and a novel synchronization construction.

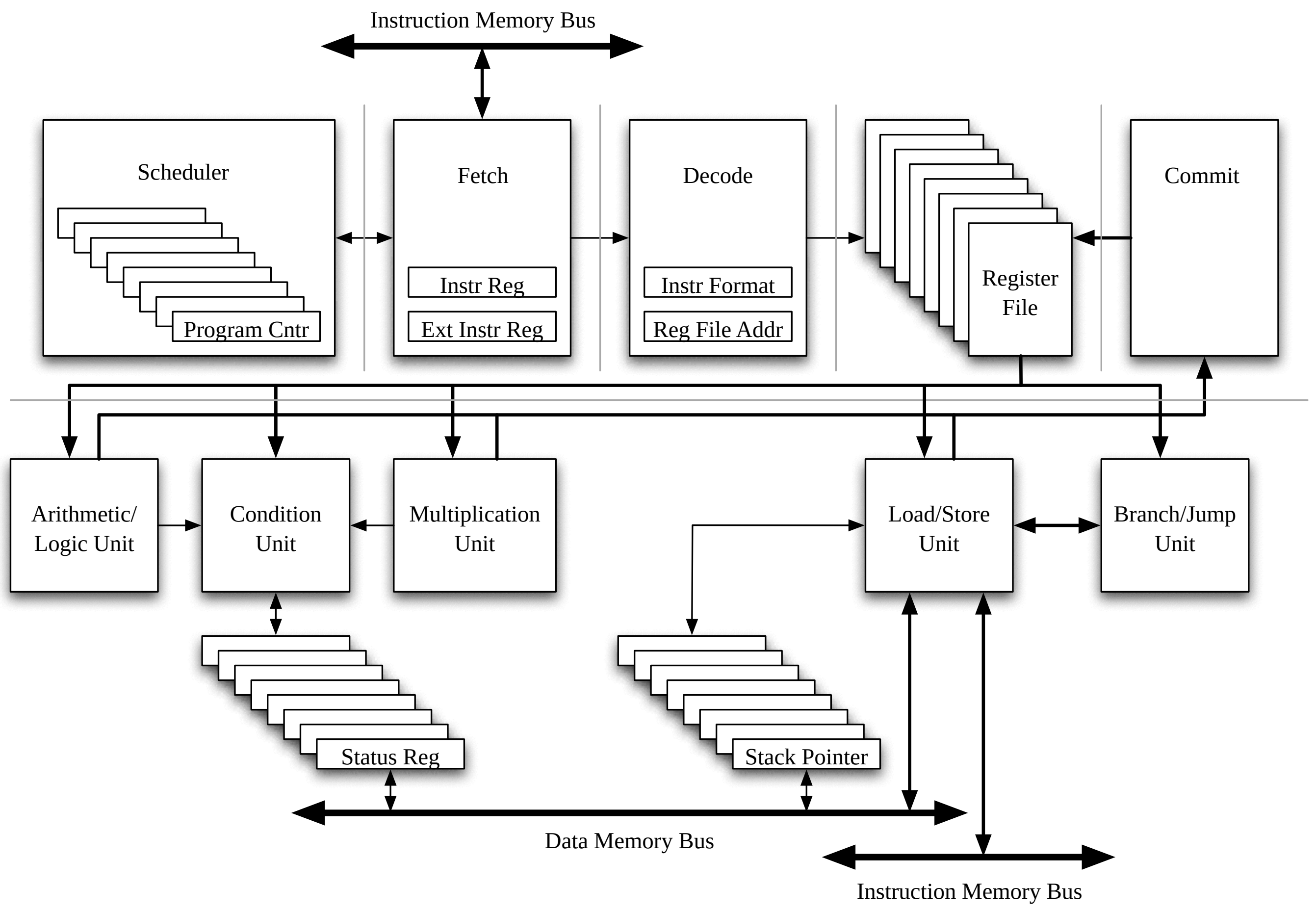


Figure 2: Functional overview of the JackKnife core components, including the division of the 6 pipeline stages.

3.1 Dynamic Interleaving

In an interleaved pipeline, instructions are selected from a different thread every cycle. This strategy provides implicit data path sharing and removes data dependencies between consecutive instructions, alleviating many costly stall conditions and providing better utilization of processor resources. Dynamic interleaving allows the scheduler to dynamically select a thread for execution based on resource availability and thread readiness. The JackKnife architecture focuses on flexibility of design and system responsiveness, selecting active threads on a round-robin basis and scheduling interrupt services threads (IST) on the next cycle.

JackKnife supports concurrent execution of any number of threads up to the current design limit of eight. Interleaved processors, like the Tera MTA, often suffer from performance degradation when the number of executing threads drops below some threshold. We have escaped this drawback through limited use of forwarding which typically reduces data-dependent stalls to a single cycle. The ability to execute single-threaded code with only slight performance degradation allows the system to respond to randomness typically exhibited in many embedded environments. While we do not directly address the need for greater than 8 concurrent threads, we expect that software-level threads could be mapped onto hardware-level threads to provide additional design flexibility.

3.2 Pipeline

JackKnife employs a six stage pipeline with extended functionality for per-stage flush and stall conditions. The event-driven nature of embedded systems means that control flow will change often and unexpectedly; a successful system should minimize the overhead of these transitions. The AVR architecture utilizes a dynamic pipeline which executes instructions in anywhere from one to four cycles, depending on instruction complexity. JackKnife inherits this dynamic behavior, but provides single-cycle support for a much greater number of instructions than the AVR.

Dynamic multi-threaded execution forces increases in pipeline complexity, requiring additional logic for recognizing data and control hazards. Managing the control logic, and all of the states needed to handle the interactions of data path stalls, flushes, interrupts, and multiple threads, can get quickly out of hand. One of the problems is that the semantics of an instruction stream should hold no matter how it is interleaved with streams from other threads. A typical state machine approach to building such a controller can quickly explode into an unmanageable number of states. The other option is to hand build a control structure that can effectively handle a particular instance of the design. This however is against our design goal of creating a configurable and extensible design (for instance, this would make changing the number of threads handled by the architecture at a hardware level very difficult). To tackle this problem we have used instruction tagging in conjunction with a novel pipeline specification methodology based around the control specification language pyPBS [9]. The non-deterministic automata (NFA) technique allows efficient controllers to be realized with minimum design effort and implementation overhead. A complete description of the methodology and synthesis language are available in [10] and [9].

3.3 Scheduler

Supporting multiple concurrent streams of execution and zero-latency interrupt response requires a custom scheduler capable of balancing system requirements. The JackKnife scheduler is a best-effort, round-robin scheduler targeted at maintaining balanced execution among threads. A round-robin scheduling policy is utilized based on the last dispatched thread, the pool of active threads, and the state of interrupts.

Scheduling can quickly become a complex process, accounting for many aspects of the system state. The JackKnife scheduler is designed with simplicity in mind, operating on a minimal set of inputs. Configuration is handled via a bank of active registers which maintain thread state and a 7-bit counter for delayed de-scheduling. The density of the AVR instruction set precludes the possibility for adding custom instructions for frequently used atomic operations, such as releasing a synchronization lock and de-scheduling a thread. Such scenarios result in race conditions, making system execution unpredictable. Delayed de-scheduling provides a solution to these issues, by allowing a thread to inform the scheduler about the number of instructions it needs to execute before being deactivated.

The addition of multi-threading to a single-threaded system poses some issues in term of thread support. Thread initialization is particularly challenging, as each processor context has no direct access to any other. To facilitate this process, the scheduler provides initialization registers which are used when a thread has yet to execute. Modification of the execution history flags and target instruction address allow any thread to change the execution flow of any other. The ability to (re)start threads at arbitrary code segments is similar to software-level threading techniques like POSIX and Java, where execution begins at a specified function/method. While these operations are universally allowed, we expect that protected thread execution and other complex scheduling can be accomplished at the software level.

System execution always begins in thread zero which enters program code at the beginning of memory. This thread is effectively used to bootstrap the initialization of other threads. Shared memory provides an easily-used communication channel for information passing between threads. This is of particular importance during thread initialization, where an appropriate address for the stack pointer must be established to avoid memory corruption. On acceptance of an interrupt, the scheduler overrides the round-robin policy, scheduling the interrupt at the soonest possible time.

3.4 Interrupts

Efficient use of system resources often necessitates the use of interrupt-driven execution models. Atmel's AVR devices support up to 35 interrupts, servicing a variety of peripherals and inputs. JackKnife provides 31 customizable, priority-based interrupt sources, allowing for the inclusion of a broad range of peripherals. Compatibility with AVR devices is maintained by executing interrupt routines out of a common jump vector located at the beginning of program memory. Code migration from AVR is as simple as assigning service threads to each interrupt source. This assignment is dynamic and configured in a set of memory-mapped control registers within the interrupt unit.

With clock speeds reaching 20MHz and minimum interrupt response times of 4 clock cycles, interrupt latency for commercially available AVRs is at least 250ns. The AVR architecture provides no hardware support for fast interrupt context switching, adding additional delay to ISR response times. AVR interrupt routines typically require prologue and epilogue as long as 17 instructions (Section 5.2), resulting in delays easily reaching 1μs. In contrast, JackKnife can achieve interrupt response times of several nanoseconds by providing zero-cycle context switching, priority scheduling, and dedicated instruction service threads (IST) [3, 21, 13, 2]. ISTs allow user code to be executed immediately, without the overhead of prologue and epilogue code. This alone has shown a nearly 50% improvement in execution time (Section 5.2).

Guaranteeing correct program execution requires that interrupt services save and restore system state prior to and following execution of interrupt routines. To enable correct program flow, interrupts trigger an implicit stack push with the servicing thread's current instruction address. This makes thread initialization important for ISTs as well as normal process threads to avoid corrupting memory. Interrupt routines typically conclude with a return instruction, causing execution flow to return to a pre-interrupt state. An IST has no meaningful previous execution point and therefore should de-schedule itself to avoid rampant execution.

While ISTs offer immediate response to interrupt events, the time to complete an interrupt routine is highly influenced by the number of concurrently running threads. Worst case execution times can be determined based on the maximum number of supported hardware threads. At the software level, the current scheduler provides the option of custom, or even dynamic, allocation of system resources to executing interrupts. For instance, a high priority interrupt could de-schedule all competing threads upon entry. At the hardware level, a priority scheduler could provide the same advantages with reduced software overhead.

3.5 Synchronization

Processor throughput is maximized when parallel processes have no inter-dependencies. However, it is often the case that several threads of execution require some combination of synchronization, communication, and data sharing. The distribution of tasks among threads makes this even more important, as it is inevitable that a coherent view of the system is needed at times. While communication and data sharing are implicitly supported by the shared memory architecture, synchronization requires specialized support to guarantee race-free execution [19, 5]. This support is frequently seen through a variety of implementations

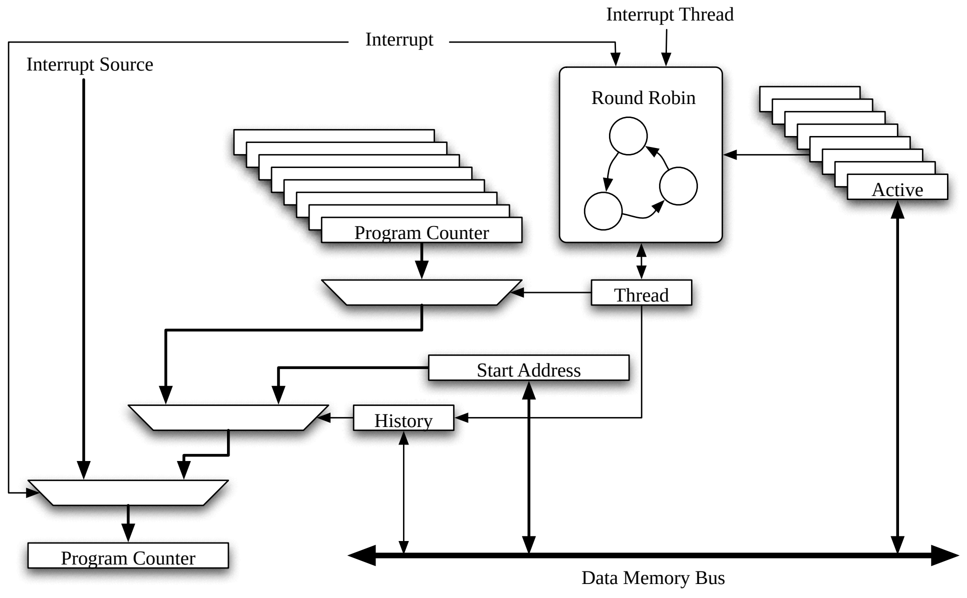


Figure 3: Functional overview of the JackKnife scheduler.

ranging from flagged memory [26, 20, 1] to dedicated instructions [26, 17]. Some of these methods aim to provide primitive constructs such as spin-locks [7, 14], while others provide more elaborate solutions.

Though there exist any number of suitable solutions, many of these techniques require custom instructions or architectural modifications that are at odds with our design goals for maintaining compatibility with existing tools and software. Instead, we have taken an approach that builds on the extensible nature of the I/O system to implement a configurable synchronization module that offers a variable number of dedicated synchronization locks. The module is a small memory with special handling of read and write requests. Obtaining one of the synchronization locks is done by performing a read request to the I/O address associated with the lock. Read requests generate an atomic test-and-set in the synchronization unit, setting the lock and returning one if the lock was previously free, and zero otherwise – effectively a 'try' operation. Locks are cleared by writing zero to the lock address.

By combining spin-locks (using the synchronization registers) with explicit scheduler control, it is possible to create more complex forms of synchronization with less overhead. Figure 4 illustrates a meeting point where all but the last thread to enter, de-schedule themselves. The final thread to enter then restarts all threads simultaneously. This example makes use of delayed de-scheduling to deactivate threads and release the synchronization lock, avoiding an otherwise potential race condition. As threads progressively deactivate themselves, processor resources are dynamically reallocated to running threads, resulting in reduced overhead and faster completion – an obvious improvement over spin-waiting. Figure 5 demonstrates a turnstile in which threads are allowed to execute in the critical region in single-file. De-scheduling all competing threads upon entry of the critical region reduces contention for resources and also removes any overhead otherwise incurred by spin-waiting.

4. MEMORY ARCHITECTURE

JackKnife implements unique contexts for all of its 8 hardware threads. Each context consists of a 32-byte register file, status register, program counter, and stack pointer. The memory architecture of JackKnife equivalently models that of the AVR, allowing for seamless code migration from existing devices. In the AVR architecture, much of the programming complexity is masked by providing a uniform interface to peripherals and memory. This allows our controller to handle a large number of different on-chip structures without requiring specialized design. Memory-mapping serves to reduce complexity stemming from otherwise necessary custom instructions and component interconnect. From an implementation standpoint, this methodology reduces the overhead of adding custom peripherals (such as our synchronization module) and control elements since interfacing requires no additional hardware.

A robust set of memory access instructions add flexibility to the AVR architecture by providing multiple ways to efficiently access both data and I/O memory. These instructions allow direct and indirect access with pre-decrement, post-increment, and displacement capabilities, as well as bit manipulation for some I/O registers. All AVR peripherals are accessible via memory-mapped I/O registers at addresses 32 - 255. Though no commercially available AVR devices use even half of this space, JackKnife allows for remapping of data memory to facilitate larger I/O memory space. While functionally disparate, I/O and data memory appear as a single contiguous memory that can be partitioned in any number of ways. This flexibility allows for straightforward customization of both on-board peripherals and memory for meeting area and functionality constraints.

The remainder of this section describes the instruction fetch and cache methodology employed in JackKnife, the register file implementation strategy, and the extensible bus architecture.

```
meeting_point(void) {
    // Spin-wait on sync lock zero
    while(!SYNC0);

    // Check if any other threads are running
    if ((ACTIVE & THREAD_GROUP_MASK)
                                    == THREAD_ID_MASK)
        // Wake all threads in group
        ACTIVE = THREAD_GROUP_MASK;

    else
        // Delayed sleep after 1 instruction
        *((char *) (ACTIVE0 + THREAD_ID)) = 3;

    // Release sync lock
    SYNC0 = 0;

    // End meeting point
}
```

Figure 4: Thread Meeting Point

```
turnstile(void) {
    // Spin-wait on sync lock zero
    while(!SYNC0);

    // Store status of threads
    unsigned char thread_status_backup = ACTIVE;

    // Sleep all other threads in group
    ACTIVE = (ACTIVE & ~THREAD_GROUP_MASK)
                                    | THREAD_THIS;

    /* Critical code segment */

    // Restore thread runnable status
    ACTIVE = thread_status_backup;

    // Release the sync lock
    SYNC0 = 0;
}
```

Figure 5: Thread Turnstile

4.1 Instruction Fetch

Many available microcontroller devices provide embedded memories for both program and data. Some devices, such as ARM, use a bootstrapping technique to move program code from non-volatile memory to volatile memory before beginning execution of the main program. JackKnife employs a hardware-level bootstrapping process to move program code to fast instruction memory. With multi-threading, one of the concerns is always the increased pressure on instruction fetch. Many of the embedded applications we target are very small, on the order of hundreds of bytes, and can be pulled completely on-chip. The ability to do single cycle access to all of memory greatly improves processor throughput when compared to the penalties incurred from instruction cache implementations and external memory accesses. In a final production design, a vast majority of the code can be stored in ROM, and a software patch memory can insure that limited software updates and bug-fixes will still be possible.

4.2 Register File

The AVR architecture provides 32 8-bit general purpose registers with support for some 16-bit operations. JackKnife implements the 8 processor contexts as a pair-wise contiguous memory. Pairing registers provides 16-bit aligned data access with minimal overhead for outputting 8-bit data. Rather than implement independent register files for each thread, a uniform memory can be optimized for speed, area, and locality with data path components. Utilizing the thread identifier as part of the register file address provides a straightforward method for accessing and updating data with very little added complexity. Rather than add complex forwarding paths, the register file provides transparent updates, effectively forwarding data for consecutive instructions. This further opens space for optimizations in the non-uniform distribution of registers to threads, and could potentially be of use in reducing the size of the register file.

4.3 Extensible Bus

As previously mentioned, the embedded domain contains many different applications that require varying types of interfaces. Because the peripheral set of a commercial (COTS) device is fixed, families of devices are typically offered with varying peripheral sets and memory sizes. It is often the case that these predetermined peripheral sets are not well suited to a particular design – included peripherals are unnecessary or do not implement the desired functionality. As a synthesizable design, JackKnife provides the ability to customize system peripherals for a given application.

Central to this capability is an extensible bus architecture that links the processor core to both data and I/O memory. Peripheral expansion can be easily accomplished while maintaining compatibility with existing AVR devices by implementing memory-mapped control registers in unused regions of the AVR I/O space. The decision to fully support a memory-mapped infrastructure allows integration complexity to be masked, and provides a uniform interface to all system components. Peripheral modules are self-contained, making them interchangeable and infinitely customizable. In addition to memory-mapped access, JackKnife provides access to its prioritized interrupt unit, allowing peripherals to request immediate servicing via one of 31 customizable interrupt sources. All peripheral modules adhere to a common bus policy in which write requests are specified as single-cycle operations and basic read requests are two-cycle operations, consisting of one cycle for address output and one cycle for data consumption. Though not advisable, the JackKnife memory controller supports the use of memory wait signals for slow memory devices. By maintaining a tight communication protocol, we aim to increase performance by reducing the number of required stall cycles.

5. IMPLEMENTATION AND SYNTHESIS RESULTS

Now that we have described the architecture in a fair amount of detail, we describe the performance and area impact as determined through synthesis, and the observed effects of multi-threading on both execution and interrupt response time. The JackKnife implementation contains roughly 4250 lines of synthesizable Verilog HDL. Data path components are all written in Verilog, while sequential controllers are specified using the pyPBS language and compiled into synthesizable Verilog.

5.1 Synthesis Results

Synopsys Design Compiler was used to synthesize several versions of the JackKnife implementation. Design Ware components are used throughout the implementation, pro-

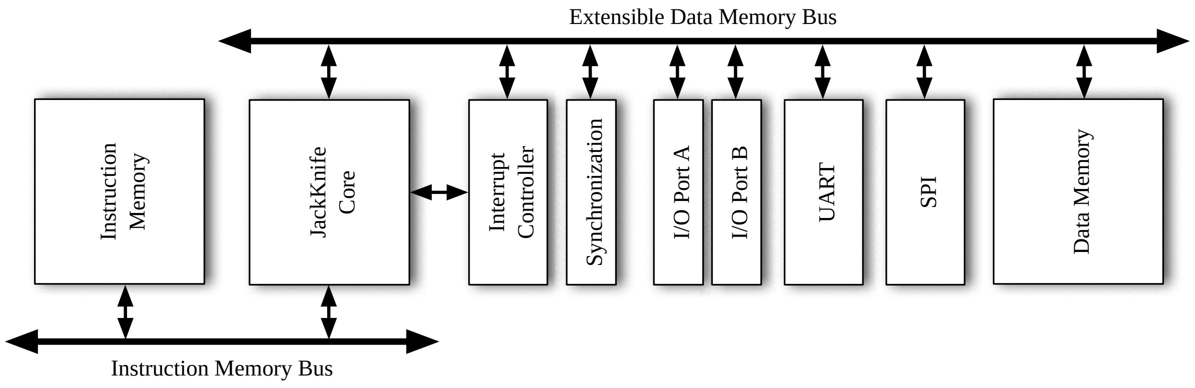


Figure 6: System-level overview of the JackKnife core and peripheral set.

viding tested and optimized components. A 0.15μm TSMC standard cell library is used for target mapping of all components. All components, including memories, are implemented in standard cells, resulting in substantially larger area and power cost than would be incurred by full-custom implementation of memory components. The TSMC cells are characteristic of typical standard cells in the technology. Conservative wiring and wire-delay models were used for performance calculations.

As we have argued throughout this paper, a multi-threaded architecture would ease the programming burden of those controlling multi-tasking embedded systems, but this can only be justified if it does not increase the size and delay of the design significantly. To show the effect of multithreading we have synthesized a single-threaded design along with multi-threaded designs with support for 2 to 8 different hardware contexts. We should point out that all of our synthesized designs run at more than 20 times higher frequency than the current best commercial AVR processors, which run at 20MHz.

The small context size of the AVR allows addition of multiple contexts at a significantly reduced cost when compared to that of a 32-bit machine. Figure 7 compares the area and clock speed trade-offs of implementing the entire JackKnife core with 1 through 8 threads. The area scales nicely with nearly a 3.5x increase from the single-thread implementation to that with 8 threads. The effective area difference between implementations consists of the size of added hardware contexts, consisting of 37 bytes of memory and supporting logic. The trade-off for multi-threading in general is shown to be minimal, with added logic complexity shown mainly in scheduling and control logic.

The clock speed is shown to remain constant across all designs with variances in design optimization accounting for any differences in maximum clock frequency. Currently, critical paths exist through the multiplication unit. This stems from single-cycle support for fractional, unsigned with signed multiplication – an operation that takes 2 cycle on the original AVR. The obscurity of such an instruction provides motivation for pipelining the multiplication unit in future implementations. While power concerns are clearly at least as important as performance, we should point out that all the performance we can extract from the machine at this stage in its design will give us slack to exploit in a full custom power optimized implementation.

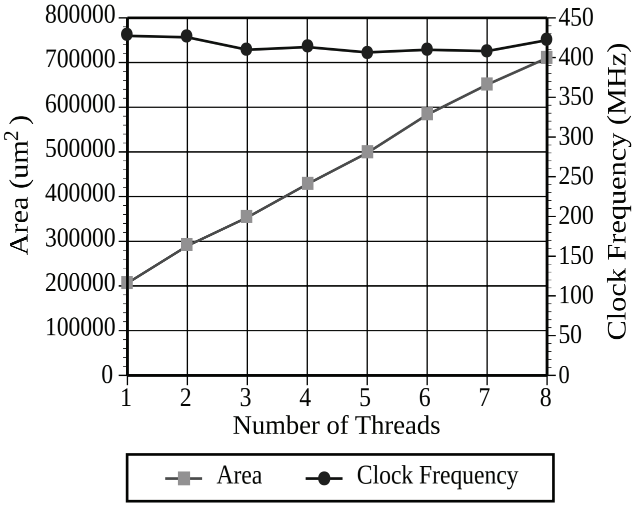


Figure 7: A plot of the Area of the JackKnife core (μm^2) and the Operating Frequency (MHz) as a function of the number of number hardware contexts supported.

To understand how the different parts of the processor are scaling with the number of contexts, Figure 8 shows the individual breakdown among processor components as they contribute to the total design area. As we have mentioned in the past, the addition of a context should increase the size of the register file and other context registers linearly, and impact some of the control and scheduling logic. At 61% of the 8-threaded design, the register file clearly dominates the total area. This is in part due to the standard cell implementation; it is important to note that the use of modern memory structures can reduce this area by as much as a factor of 12. Fabricated implementations would take advantage of modern memory structures through the use of a memory compiler, increasing performance further while significantly reducing area costs.

Looking past the size of the hardware contexts, Figure 8 shows other key components as well. The size of the decode logic remains constant across all implementations, while the control grows only slightly. This is due to a common control

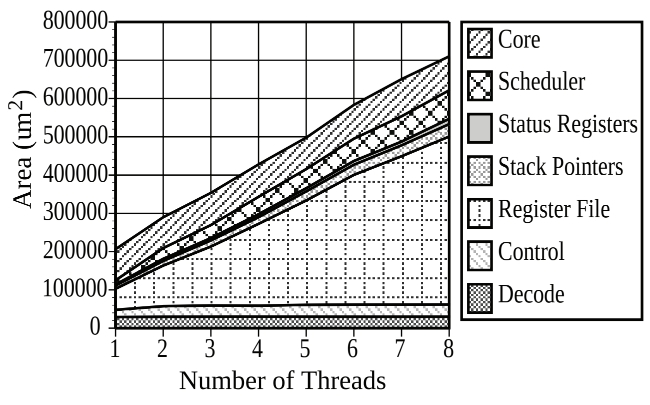


Figure 8: Area breakdown by component. As we vary the number of hardware thread contexts, this shows how each of the components of JackKnife scales in terms of required area.

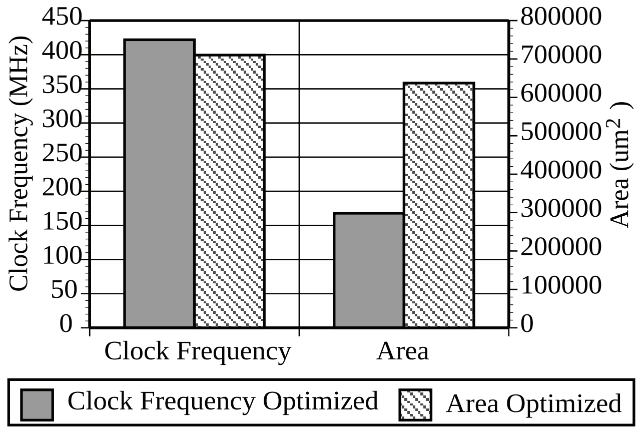


Figure 9: Constrained Optimization. This figure shows the effect of area optimized synthesis versus performance optimized synthesis.

hierarchy that is present in all implementations [10]. For single-threaded implementations, the scheduler consists of the program counter register and supporting logic. The area balance attributed to the core corresponds to the functional units and supporting glue logic. While the logic complexity of the pipeline and functional units remain constant across all designs, switching between hardware contexts comes at some expense.

Results shown thus far were synthesized using timing constraints as the primary implementation target. Figure 9 shows the synthesis trade-off for implementations targeted at area rather than performance. While the implementation targeting area is shown to be less than 90% of the size of the version strictly targeted for performance, the resulting performance cost is 2.5x. Again, the total area cost of the implementation would be greatly reduced by use of memory structures versus standard cells. Overall, our microcontroller design scales well with the number of threads and, even at 167MHz outperforms other designs in its class.

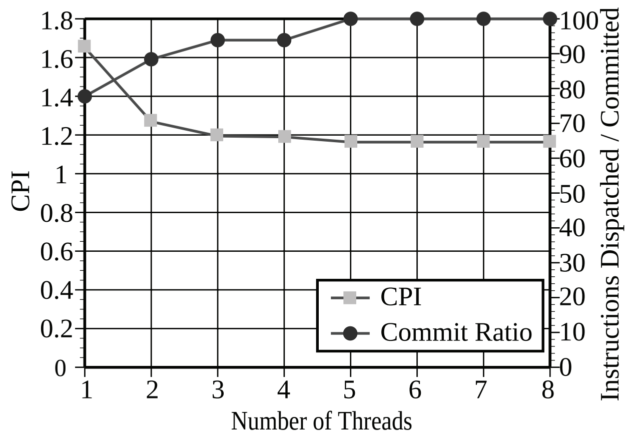


Figure 10: A plot of the CPI and ratio of dispatched to committed instructions executing a Reed-Solomon encoding algorithm.

5.2 Effect of Multi-threading

While clock speeds in excess of 400MHz provide a strong basis for our argument, they fail to convey true machine performance without some measure of the pipeline efficiency. In order to provide this measure, we simulated the JackKnife design for several test applications and characterized the performance in terms of average clocks per instruction, interrupt bandwidth, and efficiency in terms of the ratio of committed to dispatched instructions. Figure 10 plots the CPI and pipeline efficiency ratio as determined by running a Reed-Solomon encoding algorithm on 1 to 8 threads, with each thread running an independent encoding job. CPI is clearly reduced by the use of multi-threading where running more than 4 threads in parallel is shown to provide a 30% improvement compared to single-threaded execution. The committed to dispatched instruction efficiency measure shows that execution with more than 4 threads negates any waste associated with flushed in-flight instructions arising from control flow changes. In terms of power, this result is interesting as it signifies that the processor is performing 100% of its scheduled instructions, thereby making efficient use of power.

While typical CPI varies depending on the application source code, commercial AVR devices require more than one cycle to execute the majority of the instruction set. AVR devices specify a maximum CPI of 1, but like most processors, can't achieve this under typical execution – some instructions require as many as 4 clock cycles to complete and reported averages are in the range of 1.8. While the JackKnife core also has a peak CPI of 1, it typically executes at 1.16 when running more than 4 threads.

A side effect of multi-threading and the use of ISTs is the resulting advantages in terms of interrupt routine size and execution performance. By utilizing ISTs, interrupt routines can avoid the overhead associated with backing up and restoring machine state – prologue and epilogue code segments. The prologue and epilogue cannot be avoided for systems with a large number of pre-emptive interrupts however. In such systems, the eight thread limit requires that several interrupt sources map to a common thread. To

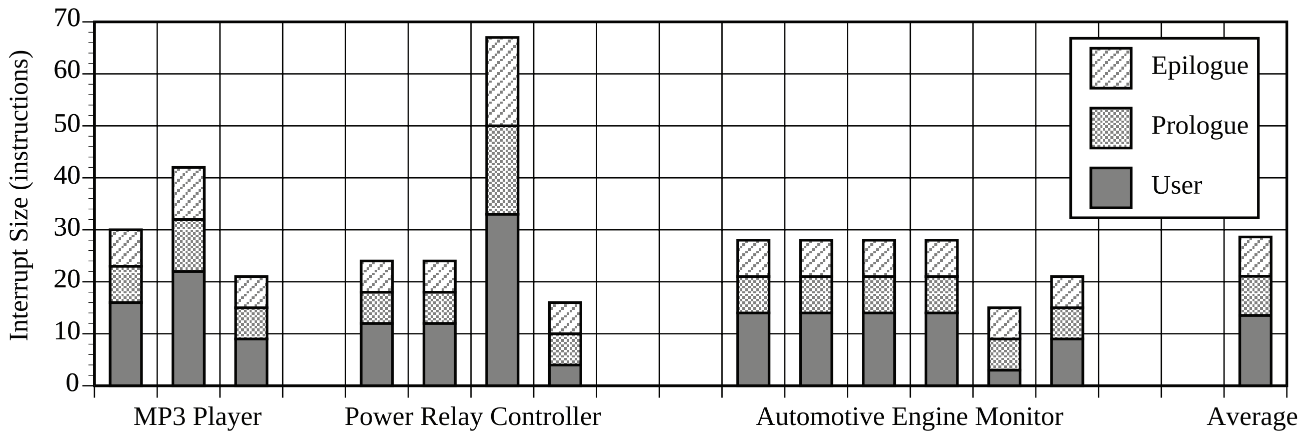


Figure 11: The sizes (instructions) of the interrupt service routines from three different AVR applications, showing the break down of prologue, epilogue, and user instructions.

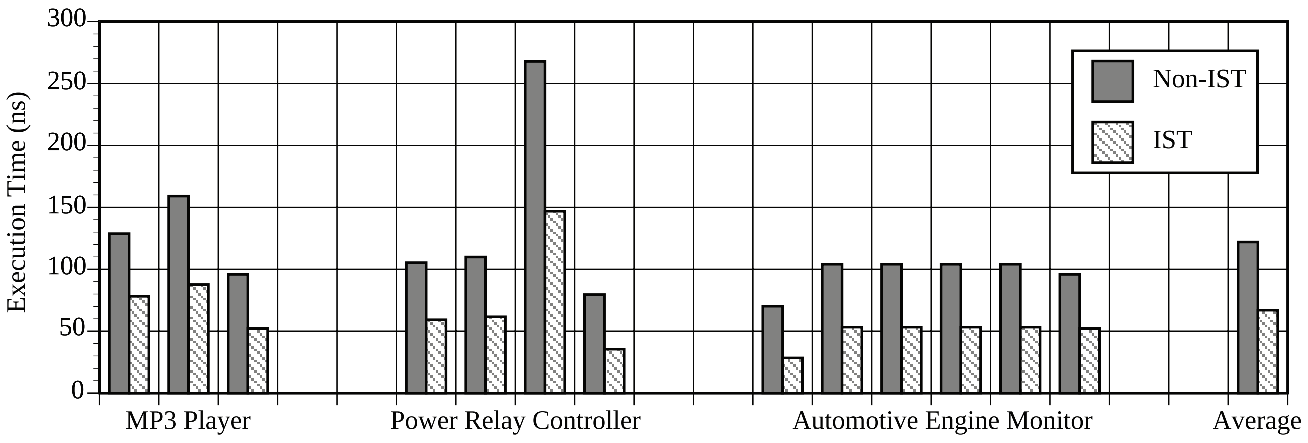


Figure 12: A comparison of execution time (ns) for running the surveyed interrupt routines in a single-threaded implementation (non-IST) versus a multi-threaded implementation employing the use of interrupt service threads.

this end, JackKnife offers a great deal of flexibility in thread mapping, allowing the software designer to make intelligent decisions based on system requirements.

We have used a set of microcontroller applications from other domains in an attempt to characterize the amount of interrupt handling code. Figure 11 surveys interrupt routine sizes for three applications: a hard disk-based MP3 player with LCD display, USB, serial, and push button interfaces; a timer and power relay controller with LCD, serial, and push button interfaces; and an automotive engine monitor with ADC, serial, and VFD interfaces. The information gathered reveals that typically 50% of the instructions in an ISR are overhead. Replacing standard interrupt routines with dedicated ISTs results in an average 50% decrease in both code size and execution time as shown in Figure 12.

In the end, the major concern is the impact of multi-threading on interrupt performance. We have argued that multi-threading benefits the designer by alleviating overhead while allowing greater bandwidth for interfacing with the external world. We have shown that the use of ISTs results in savings in code size and execution time, however we have not shown how multi-threading effects interrupt performance. Figure 13 attempts to capture the improvements in execution time resulting from modifying a typical ISR to run as an IST. It further demonstrates the average runtime of such a routine when run in parallel with at least 4 other copies. It is important to stress that all incoming interrupts are executed with single cycle response times. Running this routine in parallel on all 8 threads provides nearly 20MHz interrupt bandwidth – a 2x increase over single-threaded operation.

6. RELATED WORK

As we have described, the idea of hardware support for multi-threading is central to our design. There is a great deal of prior work from both academia and industry on multi-threading, and while a full description of all related work is not possible here, we briefly describe several related schemes as they relate to our synthesizable design.

Prior work in the area of multi-threaded architecture has

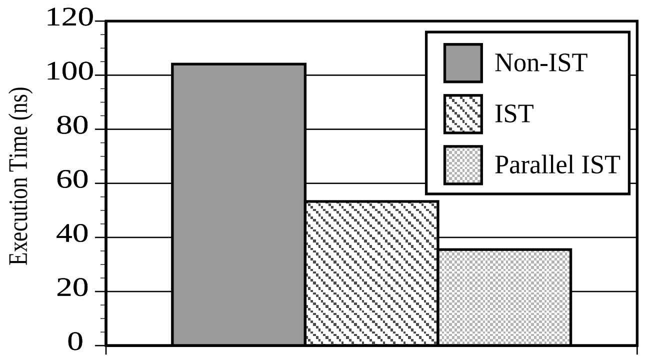


Figure 13: Comparison of execution time (ns) of a typical interrupt routine when coded for single-threaded execution, as an IST, and the average time for running multiple copies of the routine in parallel, with single-cycle response times.

primarily addressed the high-performance market, with designs targeting highly parallel large-scale applications. The Dynamic Instruction Stream Computer (DISC) [18] showed that dynamic interleaving is a viable solution to achieving better resource utilization in modern processors. The issues of synchronization, interrupt handling, and memory architecture were identified as key areas in efficient multi-threaded architectures, and approaches were presented. Interleaved multi-threading is capable of filling the vertical waste [27, 25] that occurs in conventional pipelined processor designs, but shows little advantage when utilizing more than four threads [25].

Multi-threading provides higher bandwidth than conventional processors, requiring instruction fetch and issue architectures capable of saturating the pipeline at all times [24]. Novel approaches to thread utilization [28] provide a mechanism for speculative execution, reducing the overhead of branch mispredictions. Speculative caching techniques [22] reduce the latency in instruction fetching, providing as much as a 28% improvement in performance. Novel approaches to exception handling [30] parallel the methodology employed in ISTs, removing unnecessary program serialization and allowing software techniques to achieve performance on par with sophisticated hardware.

The Komodo project [3, 21, 13, 2] has been developing a Java based microcontroller targeting real-time applications. Real-time support is guaranteed for three of the four hardware contexts, with non-real-time tasks sharing the fourth context during periods of high interrupt activity. Real-time scheduling techniques [6] bound worst case execution for multi-threaded processors, providing the capability to perform static scheduling for real-time applications. Alternatively, non-deterministic elements of the processor can be eliminated, providing predictable timing at the expense of performance. The Java microcontroller excludes cache elements and utilizes scheduling algorithms targeting shared execution, simulations of which show a 28% speed increase [3] in certain tasks when used to control an autonomous guided vehicle.

The Tera MTA [23, 4, 1, 27] is one of the more successful architectures in implementing interleaved multi-threading (IMT). Tera MTA supports multi-CPU systems where each processor can handle up to 128 concurrent threads. The architecture is optimized for execution of a large number of threads with compiler support for VLIW instructions and dependency lookahead information. Execution of small numbers of threads results in performance degradation, while only slight performance improvement is seen for variations in larger numbers of threads [20]. Single-threaded execution is impractical given that each thread may only have a single instruction in the pipeline at a time. Interrupts are supported through polling in a dedicated thread rather than supporting a preemptive interrupt architecture.

While there is a large base of theoretical work in the area of multi-threading, few physical implementations have been realized. Our work describes the architecture of a synthesizable threaded and pipelined AVR compatible microcontroller that has been mapped to standard-cell. Because this design is extensible through a simple memory mapped I/O interface, it can be easily combined on-chip with a variety of sensors to control and coordinate operation.

7. CONCLUSIONS

Many embedded applications require tight size and flexibility without sacrificing high levels of performance and interrupt bandwidth. Our multi-threaded microcontroller, JackKnife, aims to provide these features with capabilities for low-latency event handling, higher performance, and customization. Supporting interrupt service threads (IST) with zero-cycle context switching and reduced ISR overhead provides response times on the order of nanoseconds with typically 50% reduction in execution time.

We have shown that multi-threading in the embedded space, even in a pipelined machine with complex synchronization and I/O handling, is feasible and can be done with minimal overhead. Standard cell synthesis under a variety of constraints has shown that our design scales well in both performance and area, offering a range of implementations targeting power, size, and performance. From the extensible and open nature of our design, and due to the availability of mature compilers and software systems for the AVR instructions set, we believe that our design will open the opportunity to create integrated solutions not possible with off the shelf components.

8. REFERENCES

[1] G. Alverson, P. Briggs, S. Coatney, S. Kahan, and R. Korry. Tera hardware-software cooperation. In *1997 ACM/IEEE Conference on Supercomputing (CDROM)*, pages 1–16, 1997.

[2] U. Brinkschulte, C. Krakowski, C. Kreuzinger, and T. Ungerer. Interrupt service threads - a new approach to handle multiple hard real-time events on a multithreaded microcontroller. In *RTSS WIP sessions*, pages 11–15, Dec. 1999.

[3] U. Brinkschulte, C. Krakowski, J. Kreuzinger, and T. Ungerer. A multithreaded java microcontroller for thread-oriented real-time event-handling. In *1999 International Confererence on Parllel Architectures and Compilation Techniques*, pages 34–39, Oct. 1999.

[4] S. Brunett, J. Thornley, and M. Ellenbecker. An initial evaluation of the tera multithreaded architecture and programming system using the c3i parallel benchmark

suite. In *1998 ACM/IEEE Conference on Supercomputing (CDROM)*, pages 1–19, 1998.

[5] S. Carr, J. Mayo, and C.-K. Shene. Race conditions: a case study. *J. Comput. Small Coll.*, 17(1):90–105, 2001.

[6] A. El-Haj-Mahmoud and E. Rotenberg. Safely exploiting multithreaded processors to tolerate memory latency in real-time systems. In *CASES '04: Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 2–13, New York, NY, USA, 2004. ACM Press.

[7] J. R. Goodman, M. K. Vernon, and P. J. Woest. Efficient synchronization primitives for large-scale cache-coherent multiprocessors. In *ASPLOS-III: Proceedings of the third international conference on Architectural support for programming languages and operating systems*, pages 64–75, New York, NY, USA, 1989. ACM Press.

[8] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for network sensors. In *Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 93–104, 2000.

[9] G. Hoover and F. Brewer. Pypbs design and methodologies. In *Third International Conference on Formal Methods and Models for Codesign*, 2005.

[10] G. Hoover and F. Brewer and T. Sherwood. Extensible Control Architectures. In *International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*, 2006.

[11] P. Juang, H. Oki, Y. Wang, M. Martonosi, L. S. Peh, and D. Rubenstein. Energy-efficient computing for wildlife tracking: Design tradeoffs and early experience with zebranet. In *Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 96–107, 2002.

[12] R. Kling. Intel mote: An enhanced sensor network node. In *International Workshop on Advanced Sensors, Structural Health Monitoring, and Smart Sensors*, 2003.

[13] J. Kreuzinger, R. Marston, T. Ungerer, U. Brinkschulte, and C. krakowski. The komodo project: thread-based event handling supported by a multithreaded java microcontroller. In *25th EUROMICRO Conference, 1999*, volume 2, pages 122–128, Sept. 1999.

[14] C. P. Kruskal, L. Rudolph, and M. Snir. Efficient synchronization of multiprocessors with shared memory. *ACM Trans. Program. Lang. Syst.*, 10(4):579–601, 1988.

[15] T. Liu and M. Martonosi. Impala: A middleware system for managing autonomic, parallel sensor systems. In *Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 107–118, 2003.

[16] T. Liu, C. M. Sadler, P. Zhang, and M. Martonosi. Implementing software on resource-constrained mobile sensors: Experiences with impala and zebranet. In *International Conference on Mobile Systems, Applications and Services*, pages 256–269, 2004.

[17] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, 1991.

[18] M. D. Nemirovsky, F. Brewer, and R. C. Wood. Disc: Dynamic instruction stream computer. In *24th International Symposium on Microarchitecture*, pages 163–171, 1991.

[19] R. H. B. Netzer and B. P. Miller. What are race conditions?: Some issues and formalizations. *ACM Lett. Program. Lang. Syst.*, 1(1):74–88, 1992.

[20] L. Oliker and R. Biswas. Parallelization of a dynamic unstructured application using three leading paradigms. In *1999 ACM/IEEE Conference on Supercomputing (CDROM)*, 1999.

[21] M. Pfeffer, S. Uhrig, T. Ungerer, and U. Brinkshculte. A real-time java system on a multithreaded java microcontroller. In *Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, 2002 (ISORC 2002)*, pages 34–41, May 2002.

[22] E. Rotenberg, S. Bennett, and J. Smith. Trace cache: a low latency approach to high bandwidth instruction fetching. In *29th International Symposium on Microarchitecture*, pages 24–34. IEEE, Dec. 1996.

[23] A. Snavely, L. Carter, J. Boisseau, A. Majumdar, K. S. Gatlin, N. Mitchell, J. Feo, and B. Koblenz. Multi-processor performance on the tera mta. In *1998 ACM/IEEE Conference on Supercomputing (CDROM)*, pages 1–8, 1998.

[24] D. Tullsen, S. Eggers, J. Emer, H. Levy, J. Lo, and R. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *23rd Annual International Symposium on Computer Architecture*, pages 191–202, May 1996.

[25] D. Tullsen, S. Eggers, and H. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *22nd Annual International Symposium on Computer Architecture*, pages 392–403, June 1995.

[26] D. M. Tullsen, J. L. Lo, S. J. Eggers, and H. M. Levy. Supporting fine-grained synchronization on a simultaneous multithreading processor. In *Proceedings of the Fifth International Symposium on High-Performance Computer Architecture*, pages 54–58, Jan. 1999.

[27] T. Ungerer and B. Robic. A survey of processors with explicit multithreading. 35:29–63, Mar. 2003.

[28] S. Wallace, B. Calder, and D. Tullsen. Threaded multiple path execution. In *25th Annual International Symposium on Computer Architecture*, June 1998.

[29] P. Zhang, C. M. Sadler, S. A. Lyon, and M. Martonosi. Hardware design experiences in zebranet. In *International Conference on Embedded Networked Sensor Systems*, pages 227–238, 2004.

[30] C. B. Zilles, J. S. Emer, and G. S. Sohi. The use of multithreading for exception handling. In *MICRO 32: Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture*, pages 219–229, Washington, DC, USA, 1999. IEEE Computer Society.