Building Heterogeneous Reconfigurable Systems With A Hardware Microkernel

Jason Agron Dept. of Computer Science and Computer Engineering
University of Arkansas, Fayetteville, AR
jagron@uark.edu

David Andrews Dept. of Computer Science and Computer Engineering
University of Arkansas, Fayetteville, AR
dandrews@uark.edu

ABSTRACT
Field Programmable Gate Arrays (FPGAs) have long held the promise of allowing designers to create systems with performance levels close to custom circuits but with a software-like productivity for reconfiguring the gates. Unfortunately achieving this promise has been elusive. Modern platform FPGAs are now large enough to support complete heterogeneous Multiprocessor System-On-Chips (MPSoCs), however standardized design flows and programming models for such platforms do not yet exist. To achieve true software-like levels of productivity, the design flow and development environment for heterogeneous MPSoCs must resemble that of standard homogeneous systems. In this paper we present a new design flow and run-time system that enables developers to program a heterogeneous MPSoC using standard POSIX-compatible programming abstractions. The ability to use a standard programming model is achieved by using a hardware-based microkernel to provide OS services to all heterogeneous components. This approach makes programming heterogeneous MPSoCs transparent, and can increase programmer productivity by replacing synthesis of custom components with faster compilation of heterogeneous executables. The use of a hardware microkernel provides OS services in an ISA-neutral manner, which allows for seamless synchronization and communication amongst heterogeneous threads.

Categories and Subject Descriptors
C.1.2 [Computer System Organization]: Processor Architectures—Multiprocessors; D.4.7 [Software]: Operating Systems—Organization and Design, Embedded Systems

General Terms
Design, Experimentation, Performance

1. INTRODUCTION
Field Programmable Gate Arrays (FPGAs) have long held the promise of allowing designers to create systems with performance levels close to custom circuits but with the ability to reconfigure gates with software-like levels of productivity. Unfortunately achieving this promise has been elusive. While the performance of FPGA-based systems are very impressive, productivity levels for building FPGA-based systems still fall closer to those associated with designing hardware. The roots of this productivity gap trace back to the adoption of synthesis-based tool flows for embedded system and ASIC designs to program the early generations of FPGAs [18]. These design flows were appropriate for the first two generations of FPGAs, which offered modest gate densities that limited their use to co-processor accelerators that exploited gate- and data-level parallelism. To increase productivity for these use cases, researchers began investigating the use of higher-level programming language abstractions to replace hardware description languages (HDLs). This era witnessed a plethora of C-to-gates synthesis tools designed to replace hardware description languages (HDLs) [14, 23, 11, 25, 24, 21].

Although a step in the right direction, these approaches suffered a mismatch between the sequential model of the adopted programming languages and the desire to expose gate-level parallelism [6]. To address this mismatch, the languages were augmented with platform-specific pragmas and low-level constructs to allow users to expose parallelism. These approaches still required separate design flows for software and hardware, as well as custom data interfaces between the controlling CPU and co-processor circuit. Even though these approaches allowed designers to replace HDLs with more familiar software programming languages, they still relied on synthesis instead of compilation to create co-processor accelerator circuits.

Third generation FPGAs provided platforms with increased gate densities and diffused logic such as SRAM, multipliers and even processor cores within the FPGA fabric. These chips represented complete systems-on-chip components capable of supporting much more than a single co-processor accelerator. The size and capabilities of the FPGA components spurred advancements in system modeling abstractions to blur the CPU/FPGA boundary. Work such as [12, 9] provided new methods for interfacing and designing hardware components through better abstractions. Research on run-time systems for FPGAs, such as [19, 2, 13], showed how the familiar multithreaded asynchronous programming model could be used to seamlessly unify computations run-
from hardware-centric synthesis flows. The high-level application can also be used to specify and drive the numbers and types of processors required for the application [10]. Although promising, additional research is still required in three critical areas to realize this full potential. First, the reconfigurable computing community lacks the necessary diversity of heterogeneous processor cores and compilers to enable the required mix of MIMD and SIMD parallelism to replace custom circuits. Researchers are starting to address this gap with new soft-core SIMD processors [28, 31], however these cores still lack appropriate compilers to allow programmers to work with higher-level programming languages. Second, new operating system capabilities are required to enable multiprocessors with different ISAs to be integrated under a unifying virtual machine model. This issue of ISA heterogeneity is shared with the emerging manycore movement. Third, new design flows are required to allow programmers to create and link different types of binaries for the heterogeneous processors from a single high-level application.

1.1 Contributions

In this paper, we show how we extended our original hthreads hardware microkernel and created a new design flow that enables designers to specify and create heterogeneous multiprocessor systems from a single PThreads application. Hthreads provides a set of processor-agnostic system services that are equally accessible by heterogeneous processors. Importantly, the hardware microkernel approach resolves heterogeneity issues while still achieving a very tight performance envelope to meet real-time performance requirements of embedded systems. We also present a design flow that supports the creation of multiple processor-specific executables from a single PThreads application program. In our experimental results section we provide examples that show the ease in which systems with a heterogeneous mix of PowerPC and MicroBlaze processors can be targeted from a single PThreads application program. The system is designed to allow programmers to use heterogeneous processors in a generalized and transparent way. The use of a single set of standard APIs promotes portability, while still providing speedups on the heterogeneous platform. The heterogeneous platform is a prototype that consists of an embedded PowerPC 440 processor and several MicroBlaze processors in varying configurations. The system displays ISA-level heterogeneity, suitable for validating the microkernel structure and design flow. When available, more appropriate cores and compilers will be integrated into the development system. Overall, this papers contributions include:

- A hardware-based microkernel capable of supporting heterogeneous MPSoC systems composed of CPUs with different ISAs and custom hardware accelerators.
- A methodology for extending the use of POSIX-based applications to heterogeneous MPSoC environments.
- A design flow for developing heterogeneous multi-threaded code using commodity compilers and linkers.
- Experimental results of using the multi-threaded programming model to target a prototype heterogeneous MPSoC system implemented on a modern FPGA.

Figure 1: Heterogeneous Design Flow (HW + SW)

ning on both the CPU and FPGA. Figure 1 shows how our earlier hthreads operating system blurred the CPU/FPGA boundary to enable multiple synthesized hardware components within the FPGA to synchronize, communicate, and be scheduled along with software components.

The most recent generation of FPGAs, termed Platform FPGAs, have continued to follow Moore’s law in gate densities, and are now of sufficient size and complexity to support a complete multiprocessor system on chip (MPSoC). Device manufacturers, such as Xilinx, are encouraging designers to raise their traditional design focus from gates to processors as the smallest hardware design quantum [18] to handle increasing design complexity, enable processor-level parallelism, and increase productivity. This trend is now converging with the general-purpose hardware community’s move to many-core systems; both seeking to abandon complex cores that exploit instruction- and gate-level parallelism in favor of achieving increased performance through parallelism on multiple, simpler cores.

Reconfigurable MPSoCs thus promise advancements in both performance and productivity. From a performance perspective, as Moore’s law tracks the density of FPGAs, larger numbers of heterogeneous soft-core processors can be instantiated within a single chip. Adding more processors can enhance performance through increased thread-level parallelism; while adding specialized processor cores, such as array processors or extensible processors, can enhance performance through a focused acceleration of individual threads. The programmability of the FPGA provides the additional advantage of allowing designers to create a specific set of heterogeneous processors to match their applications needs.

From a productivity perspective, platform FPGAs offer the ability to target custom hardware platforms using software-centric design flows. High-level software infrastructure and operating systems can abstract away the details of FPGA-based systems. This allows programmers to use traditional software languages and compilers, and eliminates the need
2. HTHREADS HETEROGENEOUS MICROKERNEL

At run-time, applications call operating system library code. This code is typically assembly language for a specific ISA. This poses no compatibility issues for homogeneous systems. However, this does prevent a processor with a different ISA from directly invoking this library code. Inter-processor synchronization implemented using ISA-specific atomic operations is particularly problematic, as atomic operations are not uniform across different processor families [16]. Some processors, such as the MicroBlaze, have no built-in atomic operations at all. Existing heterogeneous systems such as IBM’s Cell B.E. and Intel’s EXOCHI [5, 20] resolve these differences using remote-procedure call (RPC) mechanisms that invoke services on a host processor. Although flexible, RPC mechanisms are typically implemented using heavyweight interrupt and exception mechanisms.

We developed the hthreads operating system to resolve heterogeneous incompatibilities while providing a precise real-time performance envelope for embedded systems [1, 3]. Hthreads resolved ISA incompatibilities by providing a system of processor-agnostic (ISA-neutral) thread management, scheduling, and synchronization services implemented as hardware cores. The hthreads operating system consists of 4 major hardware cores that provide OS services: the Thread Manager, Scheduler, Synchronization Manager, and Condition Variables as shown in Figure 2 [1, 3]. This modular partitioning breaks up the traditional monolithic kernel structure allowing separation of concerns between different OS service cores.

Each OS IP core fully encapsulates its internal data structures and serves as the sole interface to its internal data. This fosters explicit inter-service communications and eliminates shared data structures within the operating system itself. The basic control structure of each OS core is independent of the numbers and types of processor resources and active threads in the system. This decoupling is advantageous as each generation of manycore chips will provide performance increases through the addition of cores. The operating system must provide a framework that allows application programs to be seamlessly ported between generations as well as vendor platforms.

Each core has a simple memory-mapped interface that is accessed via traditional load/store instructions. This allows any processor that can master the bus to directly request services. This circumvents the need for slower remote-procedure call (RPC) mechanisms to provide a uniform set of efficient system services to all heterogeneous cores. Allowing each core to operate independently enables different processors to simultaneously request system services. This reduces both latency and jitter as simultaneous requests for different services do not compete for centralized services.

The internal functions of each OS IP core have also been parallelized, providing low latency system calls with minimal jitter through efficient hardware implementations. On-chip timing results are provided that were obtained on two development platforms: a Xilinx Virtex-II Pro XC2VP30 FPGA, and a Xilinx Virtex-5 FXT 70 FPGA; both using free running, cycle accurate counters.

2.1 Thread Management

The thread manager core serves as a centralized repository of thread state, regardless of the location of the thread in the system. The state of a thread indicates if the thread is used, exited, running, suspended, or ready-to-run. Thread state also includes joinable/detached status as well as parent-child relationships between threads.

The thread manager separates the concern of thread state information from the physical numbers and types of processors. This makes for a more modular OS framework through separation of thread management and scheduling policies, leading to a more scalable system in terms of both numbers and types of processors in a heterogeneous system. The state information is maintained and updated without needing to know where the thread is currently in the system. This includes threads that become ready-to-run after being unblocked from mutual exclusion primitives. Important for real-time systems, the thread manager reduces system jitter by fielding external interrupts. In a typical system, the CPU will receive interrupt requests from sources such as timers or external devices. The arrivals of these requests are outside of the control of the scheduler and interrupt the normal execution of applications. In hthreads, the thread manager fields requests, and the appropriate interrupt handler is scheduled as another thread. This puts the overhead of interrupt handling under scheduler control. Also, it makes the interrupt service routine running on the CPU very lightweight, requiring only a register read to determine the next thread and a context switch.

2.2 Scheduler

The main function of the scheduler is to determine if a new scheduling decision is required for any processor in the system. A scheduling decision can result from both implicit and explicit events. Explicitly, a thread running on a processor may exit or yield. This frees up the processor, allowing another thread that is ready-to-run to execute. Implicitly, any thread in the system can release a resource that results in the unblocking of a different thread that may preempt a currently running thread.

In general, a classic monolithic operating system scheduler is invoked for any potential scheduling decision regardless of the outcome. This can occur even when the priority of an unblocked thread will not result in a new scheduling decision. In this case, the CPU running the scheduler is in-
2.3 Synchronization

Modern processors use ISA-specific atomic instructions to perform mutual exclusion. Originally, atomic instructions such as test and set, required locking the system bus to ensure operation atomicity. This resulted in performance degradations for other processors attempting to access global memory. Bus locking can be circumvented by using a two-phase atomic operation such as the load-linked/store conditional (LL/SC, or more specifically lwarx and stwcx) instructions found with IBM’s PowerPC architecture. These instructions rely on the snoopy cache protocol to maintain atomicity between processors. The reliance on processor-specific atomic operations and the snoopy cache protocol make it difficult to perform atomic operations between different processor types [17, 16].

Our hardware synchronization core is designed to allow any type of processor to synchronize in a heterogeneous system using a single, traditional load instruction. The synchronization manager hosts a set of mutexes, in which each mutex is associated with an address offset from the base address of the IP core. The offsets of each mutex are separated by a stride indicated by the number of bits used to represent thread identifiers in hthreads. This allows an entire mutex lock/unlock command to be encoded in a single load instruction. This structure allows a processor to use a single load instruction to pass parameters to the synchronization manager in the address field, and to receive the return value of the call in the loaded value. The command contains the thread’s identifier, the mutex number, as well as the type of command. The loaded value, or return value of the command, tells the requester the result of the mutex lock/unlock operation. For a mutex lock command, this return value states whether the lock is now owned by the requester or not. If the thread did not become the owner, then the synchronization manager adds this thread to its internal blocked queue. Upon unlock operations, the synchronization manager sends the identifier of the new owner thread to the thread manager to become runnable again.

This simple protocol allows any processor that can master the bus to engage in synchronization with any other proces-
Table 2: HW Timing of Synchronization Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time (cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock Mutex</td>
<td>4</td>
</tr>
<tr>
<td>Unlock Mutex</td>
<td>5</td>
</tr>
<tr>
<td>Try Lock Mutex</td>
<td>3</td>
</tr>
<tr>
<td>Get Mutex Owner</td>
<td>3</td>
</tr>
<tr>
<td>Get Mutex Count</td>
<td>3</td>
</tr>
<tr>
<td>Get/Set Mutex Kind</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 3: Integrated Timing of Synchronization Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Avg Time (ns)</th>
<th>Std. Dev. (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock Mutex</td>
<td>1524.54</td>
<td>52.19</td>
</tr>
<tr>
<td>Unlock Mutex</td>
<td>1097.63</td>
<td>1097.63</td>
</tr>
</tbody>
</table>

The performance for the synchronization manager is shown in Table 2. This table shows that the each operation completes in five clock cycles (50 ns) or less. This level of performance is achievable with standard synchronization primitives only when processors implement caches with complex memory coherency protocols, such as snoopy caches. Table 3 shows the integrated timing results for the synchronization operations. Only the lock and unlock operations are shown, as they are the most commonly executed synchronization operations. Other synchronization operations are substantially similar. The results show that the lock and unlock commands execute in approximately 1.5 $\mu$s and 1.1 $\mu$s respectively. These averages have a standard deviation of approximately 0.05 $\mu$s and 0.02 $\mu$s respectively. The hardware overhead for these commands is 0.04 $\mu$s and 0.05 $\mu$s respectively, which means that the majority of the software side execution time is due to overhead from the bus transactions and software processing.

3. DESIGN FLOW

Traditional design and compilation flows target homogeneous systems in which all compiled code is of the same ISA. In heterogeneous systems, the presence of multiple ISAs forces the development environment to make use of multiple compilation tools, or provide an interpreter or virtual machine environment. Using an interpreter or virtual machine can lead to execution inefficiencies due to interpreter overhead [8]. To avoid this overhead we chose to create a specialized compilation technique similar to those used by IBM’s Cell and Intel’s EXOCHI [20, 7, 4] to embed heterogeneous binaries into a single executable image. The design flow described here is different in that it is able to make use of unmodified GNU compilers and linkers to produce and embed heterogeneous executables.

Our design flow allows users to work from a single application, with portions of the code destined for execution on processors with different ISAs. The build system shown in Figure 6 allows developers to work within a single application and pass thread bodies, libraries, and other support functions through separate compilers. All heterogeneous threads are defined in separate files so that they can be easily compiled for a specific target architecture. After being compiled into position-independent code, each ISA-specific executable can then be embedded into a single heterogeneous executable using a set of command line tools.

The embedding process takes an executable file, or ELF, of one architecture and embeds it into the ELF of a different architecture [7, 4]. Embedding is akin to heterogeneous linking, in which pertinent symbols within an embedded executable are made accessible to a host executable. The heterogeneous executable is flattened to preserve the memory layout of all instructions and data. Symbol table information, such as the addresses of thread start functions must be extracted as the standard GNU compilers used do not have heterogeneous linking capabilities. Additionally, threads are not first-class objects in languages such as C, as thread start functions are represented as function pointers. There-
// Embedded ELF array
unsigned char elfArray[] = {
  0xb8, 0x08, 0x00, 0x50,
  
  0x08, 0x02, 0x84
};

// Length of embedded ELF array
unsigned int elfArray_len = 1872;

// Offset for thread handle
unsigned int my_thread_HANDLE_offset = 0x000001a8;

// Thread handle
unsigned int my_thread_HANDLE =
  (unsigned int)(&elfArray) +
  (unsigned int)(my_thread_HANDLE_offset);

---

Figure 7: Example Embedded Header File

```c
#include "embed_header.h"
#define USE_HETEROGENEOUS_THREAD

void * my_thread_FUNCTION ( void * arg){
  <thread_body>
}

int main(){
  #ifdef USE_HETEROGENEOUS_THREAD
    // Creating a heterogeneous thread
    extern unsigned int my_thread_HANDLE;
    hthread_attr_init(&attr);
    hthread_attr_sethardware(&attr, VHWTI_BASE);
    status = hthread_create(
      &tid, &attr,
      (void*)my_thread_HANDLE,
      (void*)(&thread_arg)
    );
  #else
    // Creating a native thread
    status = hthread_create(
      &tid, NULL,
      (void*)my_thread_FUNCTION,
      (void*)(&thread_arg)
    );
  #endif
}
```

Figure 8: Example Main Program - Heterogeneous Thread Creation

fore each embedded ELF must contain a set of translated function pointers, called thread handles, that correspond to the heterogeneous versions of the embedded threads. These thread handles can then be used by a host application in order to refer to thread implementations targeted for specific ISAs.

A linkable C-header file is produced after flattening and symbol extraction. The file contains a pre-initialized array to hold the binary version of the flattened ELF. All thread handles are added to the file, where each thread handle is a pointer into the embedded ELF as shown in Figure 7. A program can link against this C-header file and use the thread handles to create heterogeneous threads from within a single application as shown in Figure 8. The design flow does not require modifications to be made to individual compilers, and the resulting executable does not require run-time interpretation or RPC mechanisms to invoke code on processors of different ISAs.

4. EXPERIMENTAL RESULTS

We present a series of tests that shows how our design flow and hthreads hardware microkernel allows a programmer to form a heterogeneous executable from a traditional multithreaded program with only minimal code changes. Importantly our experiments validate that complete heterogeneous parallel systems can be created, and threaded code compiled, to provide speedup without the need to pass through synthesis. The performance results do not include a speedup analysis between a custom circuit and scalar MicroBlazes. Instead, the performance results show how speedup across programmable processors is easily enhanced when additional threads are created and run in parallel across multiple MicroBlaze processors. Specialization of heterogeneous processors can provide additional speedup over thread-level parallelism alone.

4.1 Abstraction Layer

Before presenting the test cases we first outline the changes required to enable processors to replace custom circuits within an hthreads system. As shown in Figure 1 hthreads was originally developed to abstract the CPU/FPGA boundary for computations synthesized in hardware [1]. The custom circuits in hthreads required an abstraction layer that encapsulated the equivalent of a system service call, or trap to the operating system. The hardware thread interface (HWTI) serves as this abstraction layer; replacing system software libraries with finite state machine implementations.

4.1.1 Virtual HWTI

To maintain the same model for heterogeneous processors we created two new OS components: (1) the HAL, a thin software wrapper used to abstract the low-level details of the hthreads system, and (2) the virtual HWTI (V-HWTI), to replace the hardware resident finite state machines in the HWTI. The software wrapper is quite simple, transforming each high-level API call into a lower-level memory-mapped I/O command to interact with the hthreads system. Together, the HAL and V-HWTI fully encapsulate the details of the hthreads system; allowing the uniform use of system calls within a heterogeneous system. The V-HWTI is an hthreads-specific interface as it directly communicates with the OS IP cores. On the other hand, the HAL takes platform-independent APIs, and translates them into V-HWTI commands, just as a traditional OS translates system calls into trap instructions to invoke the kernel.

4.1.2 Hardware Abstraction Layer

The MicroBlaze HAL consists of two major components: V-HWTI interactions, and system call handling. The V-HWTI interactions component contains calls to initialize the V-HWTI data structures and for blocking/unblocking threads. When a heterogeneous processor is idle, the HAL acts as a small kernel that waits for a thread to be created on the heterogeneous processor. The pseudo-code of the HAL bootloop can be seen in Figure 9. When a thread is created and destined to run on a MicroBlaze, the scheduler issues a signal command to the processor’s V-HWTI. The HAL then unpacks the thread argument and function pointer from the V-HWTI, and bootstraps the thread for execution.

The HAL’s linkable high-level APIs allow heterogeneous threads to directly make system calls without the use of an RPC mechanism. The HAL is written entirely in C and
is quite portable. Each high-level API is backed by code that first converts a system call into an encoded, memory-mapped I/O command. The encoded command is then invoked, and its return value interpreted. Non-blocking system calls translate return values into user-level return values before returning control back to the calling thread. Blocking system calls interpret return values and make a decision to either continue running or to block the calling thread. If a thread should block, then it does so by waiting for a signal to enter the V-HWTI. Upon signaling, the thread unblocks and continues execution on the heterogeneous processor.
multi-threaded version of the block-based IDEA encryption algorithm. Multiple worker threads encrypt/decrypt portions of a large message in parallel. The IDEA example, along with bubblesort show how a single high-level application can be used to drive the generation of heterogeneous multiprocessor systems with different numbers and types of processor cores. These two test cases illustrate that heterogeneous systems can efficiently and use fork/join style parallelism with an appropriate run-time system.

### 4.2.3 Mailbox-based Communication

This test shows the advantage of abstraction and reuse. Both native and heterogeneous threads use the same set of APIs with no code changes required to migrate threads across the PowerPC and MicroBlaze processors as shown in Figure 12. The test creates a circular pipeline composed of parallel threads. The pipeline consists of N stages, where N is the number of worker threads plus one (the main thread). Both native- and heterogeneous-threads work together to move data through the pipeline; illustrating how heterogeneous threads cooperate, share data, and synchronize under a common run-time system and a uniform set of APIs. These tests show that increasingly complex communication constructs, such as mailboxes and bounded buffers, can be built using traditional OS primitives, and still remain usable across heterogeneous processor boundaries.

### 4.3 Performance Analysis

Our results, shown in Figure 10, clearly show how performance in a heterogeneous system can be increased through the use of more processors and the inclusion of specialized processors. In the Pi test, a single thread ran faster on the PPC compared to the MicroBlaze. Inspection of the heterogeneous executables revealed that this disparity was due to a 1.6x expansion in the number of MicroBlaze assembly instructions executed in the loop kernel when compared to the PPC. This performance disparity was overcome when greater than 2 heterogeneous worker threads were created. Figure 10 shows a 4x speedup when using 6 heterogeneous worker threads. In the divide-and-conquer tests, the single-threaded MicroBlaze performance bests the PowerPC. This results in the appearance of super-linear speedups when multiple MicroBlaze processors were used as shown in Figure 10. The root of this being that the problem is now parallelized, and each thread itself runs more quickly due to processor specialization. In the mailbox-based communication tests, each heterogeneous processor is able to execute concurrently. Thus, heterogeneous multi-threading leads to near ideal speedups in the pipeline tests as shown in Figure 10.

#### 4.3.1 Specialization

Performance was also increased by using specialized processors. In the parallel sort test, hardware FPUs were added to each MicroBlaze CPU and the code recompiled to make use of the new ISA extensions. This demonstrated the design flows ability to support a second heterogeneous ISA with specialized architectural features better matching the computations requirements. The inclusion of the hardware FPU caused the compiler to replace the software emulation code with a co-processor call transparent to the user. Interestingly, floating point emulation on the MicroBlaze is more efficient than on the PPC. This resulted from a more efficient memory interface as the MicroBlaze’s stack is located in BRAM attached to the LMB, while the PowerPC’s stack is located in DDR attached to the PLB. The lower-latency, un-arbitrated BRAM access gave the MicroBlaze better performance when accessing it’s stack. The emulation of floating point operations involved many function calls to library routines, which the MicroBlaze can handle more efficiently. Additional speedup was then gained when the multiple MicroBlaze’s were augmented with FPU-accelerators as shown in Figure 10. Faster stack access also gave the MicroBlaze the upper hand in the IDEA test, as this algorithm makes several nested calls to modulo-arithmetic functions inside of its kernel.

#### 4.4 Productivity Analysis

Custom hardware accelerators force a user to re-build (synthesize, map, and place-and-route) a complete system for every change made to the accelerator’s code. Heterogeneous systems built solely from programmable processors do not require re-synthesis, and code changes only require re-compilation. For this prototype system, re-synthesizing for a single line change of VHDL on a 2.3 GHz Intel Core 2 Duo system with 4 GB of RAM required on average 94 minutes. Re-compilation due to a change in the source code of a heterogeneous thread required on average 20 seconds (a 280x speedup). This type of rapid design space exploration is critical for application designers, allowing them to quickly explore the use of different processor configurations using the same high-level source code.

While custom hardware is often more efficient in terms of size/speed, vectorized soft processors are able to narrow the performance gap, while offering programmability and familiar development environments. The lthreads run-time system was built with this in mind, and is capable of manag-

```c
typedef struct targ{
    mailbox_t mb_start;
    mailbox_t mb_done;
} targ_t;

void* mbox_thread( void* data ) {
    targ_t * targ = (targ_t *)data;
    // Infinitely loop, read, modify write (mailboxes)
    while ( 1 ) {
        void * ptr = (void*)mailbox_read(&targ->mb_start);
        <pipeline stage code>
        mailbox_write(&targ->mb_done, (void*)result);
    }
    return NULL;
}
```

Figure 12: Mailbox Thread

<table>
<thead>
<tr>
<th>Reused Code (LoC)</th>
<th>Kernel/APIs</th>
<th>HAL</th>
<th>Base System</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>22,387</td>
<td>542</td>
<td>10,552</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>Code (LoC)</th>
<th>Reuse Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>ApproxPi</td>
<td>290</td>
<td>99%</td>
</tr>
<tr>
<td>BubbleSort(Float)</td>
<td>327</td>
<td>99%</td>
</tr>
<tr>
<td>IDEA</td>
<td>589</td>
<td>98%</td>
</tr>
<tr>
<td>Pipeline</td>
<td>554</td>
<td>98%</td>
</tr>
</tbody>
</table>

Table 4: System-Level Reuse
The ability to use a standard set of APIs from a variety of different types of processing cores fosters the ability to perform rapid design-space exploration in heterogeneous processing systems. The modular structure of the OS and HAL allow for a dramatic amount of code reuse as shown in Table 4. All APIs are uniformly available to both native- and heterogeneous-threads. Creating a thread destined for a heterogeneous processor is no different than creating a native software thread; both are coded using standard C using the same APIs. The only code changes required involve the use of a thread handle, as opposed to a traditional function pointer, and the use of an initialized thread attribute structure. These changes are done within C, are minimally invasive, and do not require a programmer to learn any new, architecture-specific APIs.

The hthreads OS is aware of all threads, both native and heterogeneous, and is able to schedule and synchronize them according to their individual attributes. Additional processors and threads can be added at will. Further, the interconnect of each processor can be customized by the system designer in order to meet the needs of an individual application without changing the high-level application code.

It is also worth noting that this productivity overview does not consider the effort to build and maintain custom hardware, nor does it consider the extra effort required to debug hardware via simulation and on-chip testing. While HLL-to-gates tools attempt to shrink this portion of the development effort, they still require a programmer to go through the long process of synthesis and place-and-route.

5. CONCLUSION

Platform FPGAs are now reaching a maturity level where they can fulfill their original promise of allowing designers to create full systems with performance levels close to custom circuits but with a software-like productivity. Achieving this promise requires creating a new infrastructure that guides us as to how to build and program such systems. We have extended the HW/SW co-designed hthreads operating system to form a wholly CPU-agnostic set of operating system capabilities. Using hthreads and our new design flow, the software portion of the system can be quickly reconfigured to create new heterogeneous processing facilities without having to undergo the slow process of re-synthesis. This enables higher programmer productivity as the function of each heterogeneous core can be changed through a fast and easy compilation and embedding process. Furthermore, this style of run-time system promotes abstraction and reuse, as all heterogeneous components interact with the same set of APIs, and the same set of operating system services.

As Moore’s law tracks the densities of next-generation FPGAs, designers will be able to integrate more processors into each chip. Current scalar processor cores can be added to increase thread-level parallelism. Specialized cores, such as extensible and array processors, can provide additional performance boosts by exploiting intra-thread data-level parallelism. Our design flow and run-time system provides the necessary framework for seamlessly mixing and matching these specialized processors with general-purpose processors. The hthreads OS provides a base on which standard programming models can be built; enabling programmers to use heterogeneous MPSoCs in much more familiar ways.

6. REFERENCES


