Hybrid Threading: A New Approach for Performance and Productivity

Glen Edwards, Convey Computer Corporation  
1302 East Collins Blvd  
Richardson, TX 75081  
(214) 666-6024  
gedwards@conveycomputer.com

Abstract: The Convey Hybrid-Core architecture combines an X86 host system with a high-performance FPGA-based coprocessor. This coprocessor is tightly integrated with the host such that the two share a coherent global memory space. In many application areas, FPGAs have shown significant speedups over what can be achieved on a conventional system. While there can be significant performance upside to porting an application to the Convey architecture, programming the FPGA-based coprocessor has always required a significant development effort. The Convey Hybrid-Threading toolset was developed to address this productivity gap, while enabling the programmer to maximize application performance.

Keywords: heterogeneous computing, hybrid-core computing, reconfigurable computing, coprocessors, accelerators, FPGAs, high-level synthesis, programming languages
**Background**
For the past several years, much of the HPC world has seen a growing gap between compute problems and compute capability. This is particularly true for the “new HPC” problems, such as those that are data-intensive rather than compute-intensive. Research in heterogeneous architectures such as GPUs and FPGAs has exploded to fill this gap.

**Convey Hybrid-Core Architecture**
The Convey Hybrid-Core architecture integrates a high-performance FPGA-based coprocessor with a standard X86 host system, and it is targeted at these new HPC apps that don’t perform well on standard architectures. The coprocessor contains four, large Xilinx FPGAs, as well as a high-performance scatter-gather memory system. The coprocessor memory pool combined with the memory on the host makes up a coherent, shared virtual memory pool that can be accessed by both the host and coprocessor. Word-addressing makes the scatter-gather memory system perform well with both sequential and random access patterns. Figure 1 below shows a block diagram of the Hybrid-Core architecture.

![Block Diagram of the Hybrid-Core Architecture](image)

The coprocessor is programmed at runtime with a “personality,” or FPGA image that executes a custom instruction set designed for the application. The personality defines the instructions that are implemented, register state and programming model. Because the personality is dynamically reloaded by the host at run time, the coprocessor can be used to accelerate many different applications.

**Parallelism for Performance**
In general, heterogeneous architectures are designed to achieve more parallelism than can be seen with a standard CPU architecture. In the case of FPGAs, this parallelism must first overcome a significant disadvantage in clock frequency. While a modern CPU is clocked in the 3GHz range, a high performance...
FPGA may be clocked at 300MHz. So in order to accelerate an application, an FPGA-based system must first overcome a 10x frequency disadvantage.

The way it overcomes this frequency gap is through parallelism. A scalar CPU executes primitive instructions, such as load, store, add or multiply, one per clock cycle. The operations are fixed, data types are fixed, and the number and type of registers are fixed. At any one point in time, many of the CPU resources may be idle because they are not needed by a particular application.

**Wide Open Spaces**

An FPGA can be programmed to be application-specific because it can be reprogrammed at runtime. The programmer has the luxury of dedicating 100% of the device to the application being run. FPGAs give the programmer the ultimate flexibility in how an algorithm is mapped to hardware. What programming model will be used? SIMD or MIMD? Streaming or multi-threaded? How is data reused to minimize memory access? How many bits are actually needed to represent the numerical range and precision required by the application? The FPGA consists of a number of resources that are available to the user—logic elements (LUTS), registers, RAMs and DSPs. Because the FPGA is dynamically reprogrammable, the user can program the FPGA to do exactly what is required by the application.

**With Great Freedom Comes Great Responsibility**

But there are many challenges associated with programming of FPGAs. Most programmers know C/C++, but FPGAs are typically programmed in a Hardware Description Language—Verilog or VHDL. Further, even for designers who are proficient in Verilog or VHDL, mapping a complex algorithm to hardware is a very slow and tedious process. The designer must define what happens in each module on each clock cycle, how each module communicates to other modules, and how much logic is implemented in a single clock cycle. The hardware design process is long and slow for several reasons:

- Architecting a hardware solution take special expertise (too much freedom?)
- HDL design is tedious
- Debug is difficult
- Timing closure is time-consuming

Because of the long design cycle, architectural exploration is very expensive. By the time a typical design cycle of nine to twelve months is complete, the designer may have learned new ways to approach the problem that, if implemented, could yield a significant improvement in the overall application performance. But after a long design, test and debug cycle, the conclusion that the design is “good enough” usually prevails, leaving performance on the table in the interest of moving on to a different problem.
The Hybrid Threading Approach

The Hybrid-Threading (HT) approach was designed specifically to address the challenges listed above. It was designed based on the following goals:

- **Decrease the time it takes to develop a custom personality**: Productivity is commonly cited as the primary roadblock to FPGAs being adopted for mainstream High Performance Computing. The HT tools are designed to decrease the time required to port an application to the Convey architecture.

- **Increase the number of people who can develop custom personalities**: While some hardware knowledge is still required of the developer, the HT tools make it easier for a non-expert to be productive on the Convey system.

- **Don’t sacrifice kernel performance**: The HT tools were designed with a bottom-up approach, starting with the low-level hardware in the FPGA and layering functionality on top to abstract away the details. The programming interface is greatly simplified, but the user still maintains the flexibility required to achieve the maximum performance from the FPGA using features such as pipelining and module replication.

- **Maximize application performance**: Kernel performance alone doesn’t matter. What matters is how much the application performance is improved. Maximizing application performance means efficiently partitioning the problem to use the best tool for the job (host or coprocessor), and making the best use of each resource (i.e. multithreading on the host processor, overlapping processing and data movement, etc.).
Hybrid Threading Architecture

The Hybrid-Threading toolset is a framework for developing applications. It is made up of two main components: the HT host API, and the HT programming language for programming the coprocessor. The programmer starts by profiling an application to determine which functions in the call graph are suitable to be moved to the coprocessor. Figure 2 below shows an example call graph. The main function and function fn5 remain on the host, while functions f1, fn2, fn3 and fn4 are moved to the coprocessor. Each of these functions is effectively programmed as a single thread of execution using the HT programming language, and the tool automatically threads each function to achieve desired parallelism.

![Application Call Graph](image)

**Figure 2. Application Call Graph**

Execution of an HT program starts with a program running on the host processor. The host processor begins a dispatch to the coprocessor by first constructing the Host Interface (HIF) class. This loads the personality into the FPGAs, allocates communication queues in memory, and starts the HIF thread on the coprocessor. This thread then spins waiting for calls or messages to be sent from the host.

Once the HIF class has been constructed, the program can allocate units on the coprocessor. A unit is made up of one or more modules, and a unit is replicated by the HT tools by a factor specified by the
user. The HIF class provides a member function that returns the number of units available. The host then communicates with units independently through call/return and messaging interfaces. The units are spread across AE FPGAs, but the number of AEs is abstracted from the user, and only the number of units is visible to the application.

A module implements a function using a series of instructions defined by the user. A module can communicate with the host and with other modules through call/return and messaging interfaces. A module can have one or more threads, and the threads are time-division multiplexed on the module so that on each clock cycle, one instruction is executed by one of the threads.

When a multi-threaded module is called, a thread is spawned which adds an instruction to the new instruction queue. On each cycle, an instruction is selected from one of the instruction queues for execution. The instruction execution is then pipelined over multiple stages so that the HT infrastructure can access thread state which may be stored in RAMs requiring multi-cycle accesses. On the execute cycle, the custom instruction programmed by the user is executed. The instruction can modify variables, read or write memory, call another function, or communicate with other modules or the host using messages. Following the execute cycle, variables are saved back to RAMs so they can be accessed on subsequent instructions executed by the thread. Figure 3 below illustrates HT instruction execution.

![Figure 3. HT Instruction Execution](image-url)
**Host Interface (HIF)**

The HT Host Interface (HIF) enables communication between the host application and personality units. The host communicates to each unit through memory-based queues:

- Control queues pass fixed-size (8B) messages
- Data queues pass variable-sized messages

The host API provides three types of transfers which are used to communicate between the host and coprocessor:

- 8B message: SendHostMsg(msg), RecvHostMsg(msg)
- Call/return with arguments: SendCall_func(args), RecvReturn_func(args)
- Streaming data: SendHostData(size, &data), RecvHostData(&data)

The HIF queue structures are replicated for each unit, so that pending calls, returns or messages to or from a unit cannot block progress of another unit. The host interface is shown below in Figure 4.

**HT Programming Language**

Units are programmed using the HT programming language, which is a subset of C++ and includes a configurable runtime that provides commonly-used functions. The user writes C++ source for each module, as well as a hybrid threading description file (HTD) which describes how the modules are called and connected, and which provides declarations for module instructions and variables. The diagram in Figure 4 shows how modules are connected inside a unit, and how the unit is connected to the host through the host interface (HIF).

**Figure 4. HT Unit and Modules**
The HTD file is used to generate a configurable runtime for each module. In the example above, Mod2 calls Mod3 and can access main memory. In the HTD file, the user defines these capabilities, and based on this file, the Hybrid Threading Linker (HTL) generates calls accessible from Mod2 for these functions:

- SendCall_Mod3()
- ReceiveReturn_Mod3()
- ReadMem_<var>(addr)

Because the programmer explicitly defines the interconnections and capabilities for the module, the HT tools generate only the infrastructure needed by the module, and therefore avoid wasting space on unneeded logic.

**Variables**
The HT programming language supports three types of variables: private, shared and global. All of these variables are “state,” meaning they are saved by the infrastructure at the end of each execution cycle. HT variable types are illustrated in Figure 5 below.

- Private variables are private to a thread. They can be modified by the thread itself, or initialized by an argument passed on the function call. The HT tools automatically replicate private variables by the number of threads specified in the module. Private variables can be scalar, one or two-dimensional.
- Shared variables are accessible by all threads in a module. They can be modified by a thread, or by a return from memory. They can be scalar variables, memories or queues.
- Global variables are accessible by all modules in a unit. They can be scalar variables or memories, and they can be modified by threads or by reads from memory.
HT Instructions

A module is written as a sequence of instructions. An instruction is one or more operations that will execute within a single clock cycle and typically consists of 10-50 lines of C++ code. Based on the capabilities defined in the HTD file, HTL generates calls to be used in the HT instructions:

- Program execution: HtRetry(), HtContinue(), HtPause(), HtResume()
- Call/Return: SendCall_<func>(), SendCallFork_<func>(), RecvReturn_<func>(), ...
- Communication: SendHostMsg(), SendHostData(), SendMsg(), RecvMsg(), …

At the beginning of each instruction, needed resources must be checked for availability. For example, if an instruction does a write to memory, a WriteMemBusy() call checks that the write buffer has space for the transaction. If the busy check fails, the program can re-execute the same instruction using HtRetry(), or it can do another operation. The simulation environment includes runtime assertions to ensure resources are checked for availability before being used.

After the operations are performed, the instruction calls a function which indicates what the thread should do on the next cycle of execution. Some examples are:

- HtContinue(CTL_ADD) – continue executing at the CTL_ADD instruction
- HtRetry() – retry current instruction (state is unmodified)
- SendReturn_<func> - return from the function call (kills this thread)
- ReadMemPause(ADD_ST) – pause this thread until all outstanding reads are complete
**Hybrid-Threading Tool Flow**

The Hybrid-Threading tool flow is made up of two primary programs: HTL (Hybrid Threading Linker) and HTV (Hybrid Threading Verilog generator). The programmer provides an HT description file (HTD) which defines the modules, declares variables and describes other module capabilities. This file is read by HTL to produce the configurable runtime libraries used in the custom instructions source files (*_src.cpp). The custom instruction source files are then compiled into an intermediate SystemC representation to build a simulation executable. At this stage, the programmer can easily debug the design using printf, GDB, assertions, etc. Because the SystemC simulation is cycle-accurate, it also allows for rapid design exploration.

![Diagram of the HT tool flow](image)

**Figure 6. Hybrid-Threading Tool Flow**

After verifying the design in simulation, the HTV tool is used to generate Verilog. The Verilog can be simulated using an HDL simulator, if desired, before building the FPGA using the Xilinx ISE design tools. The resulting bitfile is packaged into a personality to be run on the Convey Hybrid-Core server. A diagram of the HT tool flow is shown in Figure 6.

**Debugging and Profiling**

In addition to using standard debugging methods during simulation (printf, GDB, etc.), the HT tools provide an HtAssert() function. In simulation, this acts as a normal assert() call, which aborts the
program if the expression fails. But when the Verilog is generated for the personality, HTV also generates logic for the assertion so that it also exists in the actual hardware design. If an assertion fails, a messages with the instruction source file (*.src.cpp) and line number is sent to the host and printed to standard error. Finding a problem at the source due to an assertion is significantly easier than trying to trace back from a symptom such as a wrong answer, and because the HtAssert() call is built in to the HT infrastructure, no special debugging tools are required to use it. The user can choose to globally enable or disable asserts as the design moves from functionality verification to production use.

HT also provides a profiler to provide instruction and memory tracing, program cycle counts, and other information that is useful in optimizing performance of a design. This helps the programmer to explore architectural changes and identify performance bottlenecks before running on the actual hardware. Figure 7 shows the performance monitor output (HtMon.txt) from the vector add example described in the next section.

```
Total Clock Count: 608016

HW Thread Group Busy Counts
  CTL       Busy: 475137

Total Memory Read / Write: 201269 / 100624
Memory Read Latency Min / Avg / Max: 69 / 100 / 479

Memory Read / Write Counts
  HIF Read / Write: 22187 / 624
  CTL Read / Write: 0 / 0
  ADD Read / Write: 200000 / 100000

Active HtId Counts
  ADD Avg / Max: 57 / 128

Module Instruction Valid / Retry Counts
  CTL                  235681 / 35633
  ADD                  464500 / 64500

Module CTL Individual Instruction Valid / Retry Counts
  CTL_ENTRY            16 / 0
  CTL_ADD              135649 / 35633
  CTL_JOIN             100000 / 0
  CTL_RTN              16 / 0

Module ADD Individual Instruction Valid / Retry Counts
  ADD_LD1              126523 / 26523
  ADD_LD2              126830 / 26830
  ADD_ST               111147 / 11147
  ADD_RTN              100000 / 0
```

Figure 7. HT Performance Monitor Output

**Vector Add Example**

A simple vector add example illustrates the use of the Hybrid Threading toolset. This example adds two arrays of 64-bit integers into a third array and returns the sum reduction of a3.
Of primary concern on a problem like this is how to maximize memory bandwidth, since it will clearly be a memory-bound problem. In the Hybrid-Core architecture, maximizing memory bandwidth means doing two things:

1. Send a memory request every clock cycle from every memory port
2. Keep enough memory requests outstanding to overcome the memory latency

In the Vadd example implementation, both of these are largely handled by tools. Because the units are automatically replicated, and because each unit is connected to a memory port, all memory ports are used. The main function divides the work across units by providing an offset (unit number) and a stride (number of units). Threads are used to keep many transactions outstanding in the memory subsystem. A control function “CTL” calls a function “ADD” asynchronously to spawn threads, and each ADD thread processes a single array element. Figure 8 shows the call graph for the Vadd implementation in HT.

```c
for (i = 0; I < length; i++) {
    a3[i] = a1[i] + a2[i];
    sum += a3[i];
}
return (sum);
```

**Figure 8. Vector Add Call Graph**

The HT source code for the CTL module is shown in Figure 9. The CTL module has a single thread of execution, and its job is to spawn threads to process individual elements of the array. The entry point instruction is CTL_ENTRY, which initializes variables and then continues to the CTL_ADD instruction. The CTL_ADD instruction first checks if the call to the add function is busy—if it is, it retries the instruction on the next cycle using HtRetry(). Threads are spawned, one for each array index, using the asynchronous SendCallFork_add() function call. When threads have been spawned to process all array elements, the RecvReturnPause_add() function puts the control thread to sleep until all calls have returned. Returning threads are joined in the CTLJOIN state, where the sum is accumulated in a shared variable (prefixed by S_). When all threads have returned, the control function returns the sum in the CTL_RTN state.
#include "Ht.h"
#include "PersAuCtl.h"

void CPersAuCtl::PersAuCtl() {
    if (PR_htValid) {
        switch (PR_htInst) {
            case CTL_ENTRY: {
                S_sum = 0;
                P_result = 0;
                HtContinue(CTL_ADD);
            }
            break;
            case CTL_ADD: {
                if (SendCallBusy_add()) {
                    HtRetry();
                    break;
                }

                if (P_vecIdx < S_vecLen) {
                    SendCallFork_add(CTL_JOIN, P_vecIdx);
                    HtContinue(CTL_ADD);
                    P_vecIdx += P_vecStride;
                } else {
                    RecvReturnPause_add(CTL_RTN);
                }
            }
            break;
            case CTL_JOIN: {
                S_sum += P_result;
                RecvReturnJoin_add();
            }
            break;
            case CTL_RTN: {
                if (SendReturnBusy_htmain()) {
                    HtRetry();
                    break;
                }

                SendReturn_htmain(S_sum);
            }
            break;
            default:
                assert(0);
            }
        }
    }
}

Figure 9. CTL Module Source

The source code for the ADD function is shown in Figure 10. The entry point is the ADD_LD1 instruction, which loads an element from array a1 and continues to ADD_LD2. The ADD_LD2 instruction loads an element from array a2. The thread cannot execute the next instruction ADD_ST until the read data has returned from memory, so the ReadMemPause(ADD_ST) function is called to put the thread to sleep until the reads have returned. The read data is stored in shared variable memory structures, one for each operand, and indexed by the thread ID. When the data has returned from memory, the ADD_ST instruction is automatically scheduled for execution by the HT infrastructure.
The ADD_ST instruction adds the operands, stores the result, and then puts the thread to sleep until the write has completed to memory using WriteMemPause(). Finally, the ADD_RTN instruction returns the sum to the CTL module. The sum is stored in a private variable (prefixed by P_), so that each thread’s sum is preserved.

The latency to memory on the Convey Hybrid-Core architecture is relatively high, on the order of 100 clock cycles. As a result, the threads are sleeping the majority of the time waiting for memory. But because there are enough threads per ADD module to keep many requests outstanding (128 threads in the case of this design), the design is able to maximize memory bandwidth.
```c
#include "Ht.h"
#include "PersAuAdd.h"

void CPersAuAdd::PersAuAdd() {
  S_op1Mem.read_addr(PR_htId);
  S_op2Mem.read_addr(PR_htId);
  if (PR_htValid) {
    switch (PR_htInst) {
      case ADD_LD1: {
        if (ReadMemBusy()) {
          HtRetry();
          break;
        }
        // Memory read request
        MemAddr_t memRdAddr = S_op1Addr + (P_vecIdx << 3);
        ReadMem_op1Mem(memRdAddr, PR_htId);
        HtContinue(ADD_LD2);
        break;
      }
      break;
      case ADD_ID2: {>
        if (ReadMemBusy()) {
          HtRetry();
          break;
        }
        // Memory read request
        MemAddr_t memRdAddr = S_op2Addr + (P_vecIdx << 3);
        ReadMem_op2Mem(memRdAddr, PR_htId);
        ReadMemPause(ADD_ST);
        break;
      }
      break;
      case ADD_ST: {
        if (WriteMemBusy()) {
          HtRetry();
          break;
        }
        P_result = S_op1Mem.read_mem() + S_op2Mem.read_mem();
        // Memory write request
        MemAddr_t memWrAddr = S_resAddr + (P_vecIdx << 3);
        WriteMem(memWrAddr, P_result);
        WriteMemPause(ADD_RTN);
        break;
      }
      break;
      case ADD_RTN: {>
        if (SendReturnBusy_add()) {
          HtRetry();
          break;
        }
        SendReturn_add(P_result);
      }
      break;
      default:
        assert(0);
    }
  }
}
```

Figure 10. ADD Module Source
Code Reduction
The vector add example is also implemented in Verilog as an example for the Personality Development Kit. It has the same number of design units and similar functionality to the version implemented in HT. It therefore serves as a good comparison to measure the reduction in source code. Table 1 shows the difference—a ninety percent reduction in code—between the two implementations.

<table>
<thead>
<tr>
<th>Verilog/PDK Source</th>
<th>Hybrid Threading Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bytes</td>
<td>Source File</td>
</tr>
<tr>
<td>2639</td>
<td>cae_clock.v</td>
</tr>
<tr>
<td>3207</td>
<td>cae_csr.v</td>
</tr>
<tr>
<td>22891</td>
<td>cae_pers.v</td>
</tr>
<tr>
<td>2616</td>
<td>instdec.v</td>
</tr>
<tr>
<td>10186</td>
<td>vadd.v</td>
</tr>
<tr>
<td><strong>41539</strong></td>
<td><strong>3745</strong></td>
</tr>
</tbody>
</table>

Table 1. Source Code Comparison

Conclusion
The Hybrid-Threading approach is designed to significantly reduce the time it takes to design and debug a custom personality for the Convey Hybrid-Core architecture. The host interface provides a simple API for communication between the host and coprocessor, and the HT Programming Language removes much of the tedium associated with developing custom FPGA logic. Debugging and profiling tools aide the developer in verifying the design and tuning performance.

Glen Edwards
Glen Edwards is the Director of Design Productivity Technology at Convey Computer. Prior to joining Convey, he was a hardware design engineer at Hewlett-Packard, where he designed system boards and FPGAs for HP's high-end scalable servers.

Xilinx and ISE are registered trademarks of Xilinx. Intel is a registered trademark of Intel Corporation. Convey Computer, Hybrid-Core, Hybrid-Threading and HT are trademarks of Convey Computer Corporation.