Thread Warping: A Framework for Dynamic Synthesis of Thread Accelerators

Greg Stitt
Department of Electrical and Computer Engineering
University of Florida
gstitt@ece.ufl.edu

Frank Vahid
Department of Computer Science and Engineering
University of California, Riverside
vahid@cs.ucr.edu
Also with the Center for Embedded Computer Systems, UC Irvine

ABSTRACT
We present a dynamic optimization technique, *thread warping*, that uses a single processor on a multiprocessor system to dynamically synthesize threads into custom accelerator circuits on FPGAs (field-programmable gate arrays). Building on dynamic synthesis for single-processor single-thread systems, known as warp processing, thread warping improves performances of multiprocessor systems by speeding up individual threads and by allowing more threads to execute concurrently. Furthermore, thread warping maintains the important separation of function from architecture, enabling portability of applications to architectures with different quantities of microprocessors and FPGA—an advantage not shared by static compilation/synthesis approaches. We introduce a framework of architecture, CAD tools, and operating system that together support thread warping. We summarize experiments on an extensive architectural simulation framework we developed, showing application speedups of 4x to 502x, averaging 130x compared to a multiprocessor system having four ARM11 microprocessors, for eight benchmark applications. Even compared to a 64-processor system, thread warping achieves 11x speedup.

Categories and Subject Descriptors
C.3 [**Special-Purpose and Application-Based Systems**]: *Real-time and embedded systems.*

General Terms
Performance, Design.

Keywords
Synthesis, FPGA, threads, multi-core, dynamic synthesis, warp processing, thread warping, just-in-time compilation.

1. INTRODUCTION
Modern processing architectures are increasingly multi-core, containing multiple processors (cores) on a single chip [13][15], even near 100 [18]. Multi-core devices are commonly connected on boards or backplanes to form even larger multiprocessor systems [13][26]. The mainstreaming of multiprocessor architectures increases the number of applications written using multiple threads, which can better utilize multi-core parallelism compared to single-threaded applications.

Field-programmable gate array (FPGA) devices are being incorporated into many multiprocessing systems, by multiprocessor vendors [5][14][26], FPGA vendors [29][30], and third-party vendors [6]. Such incorporation is due to FPGAs providing several-order-of-magnitude speedups for many applications ranging from embedded systems [11][21][25] to supercomputing [6]. Yet, a widely recognized barrier to wider FPGA utilization is the challenge of programming mixed multiprocessor/FPGA systems, with extensive language and tool research seeking to overcome the barrier [8][9][11][16][21].

Meanwhile, multiprocessor architecture researchers have proposed a paradigm in which one processor performs optimizations that benefit other processors [19][31]. Such optimizations might include detecting and re-compiling (with high optimization) critical code regions, just-in-time compiling critical regions to a processor's native instruction set (e.g., VLIW rather than x86), scheduling threads, scaling voltages, etc. Separately, we previously developed *warp processing*, which uses an on-chip processor to dynamically remap critical code regions from processor instructions to FPGA circuits [21] using runtime synthesis. That work showed that aggressive decompilation can often recover enough high-level information (e.g., loop, arrays, subroutines) from a binary to yield synthesized circuits competitive with those synthesized from source code.

We combine these concepts, by proposing a new dynamic multiprocessor optimization technique, *thread warping*, that uses one processor to synthesize threads into circuits on FPGAs. Despite the fact that modern synthesis tools have long execution times, we show that order-of-magnitude application speedups can be obtained for numerous compute-intensive benchmark applications. Our contribution consists of integrating existing and new CAD techniques into a framework capable of dynamically synthesizing thread accelerators, allowing an operating system to schedule threads onto both microprocessors and custom circuits.

This paper is organized as follows. Section 2 provides a technique overview. Section 3 describes on-chip CAD tools for synthesizing thread accelerators. Section 4 discusses operating system support. Section 5 presents experimental results.

2. OVERVIEW
Figure 1(a) overviews thread warping for a multithreaded application, which initially creates threads to execute function *f()*, as shown in step 1 of the figure. Because the number of threads

CODES+ISSS'07, September 30–October 3, 2007, Salzburg, Austria.

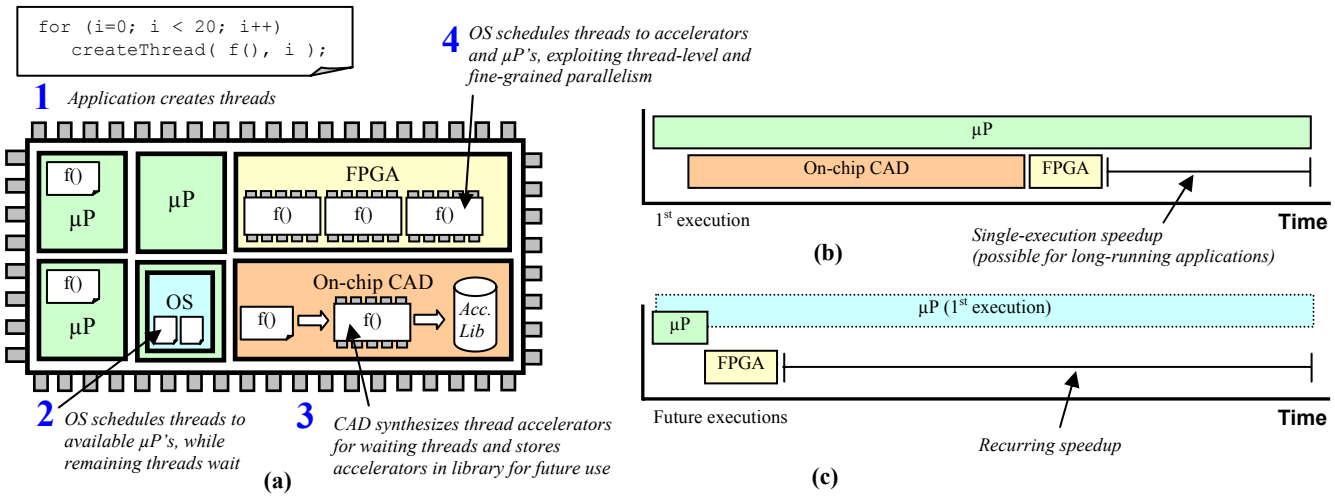

Figure 1: Thread warping: (a) architecture and overview, (b) usage when application runtimes greatly exceed synthesis times, as for scientific computations that run for hours or days, (c) usage when applications repeat, in which case synthesized results can be stored in a library and retrieved when needed.

exceeds the number of processors, the operating system places threads in a queue as they await an available processor, as shown in step 2.

Our framework monitors this queue, analyzes waiting threads, and utilizes on-chip CAD tools to create custom accelerator circuits for the *f()* function (step 3). After some time (32 minutes using existing commercial synthesis tools for our benchmark set), the CAD tools finish mapping the accelerators onto the FPGA. Assuming the application has not finished, as in Figure 1(b), the operating system (OS) begins scheduling threads onto both FPGA accelerators and microprocessor cores (step 4), resulting in a speedup for a single execution. Single-execution speedups are possible for long-running applications from domains such as scientific computing.

Of course, for many applications, 32 minutes of synthesis may exceed the application's execution time. For shorter applications, such as embedded systems, the framework caches accelerators in a non-volatile library for future executions, thus amortizing the synthesis time over the repeated executions, as in Figure 1(c), resulting in an average speedup of 130x compared to four ARM11 microprocessors for our benchmarks. Ultra-efficient synthesis techniques [20][21], possibly coupled with fast, synthesis-oriented FPGAs [1], may help make single-execution speedups possible for more applications.

Although the architecture in Figure 1 utilizes four cores and an on-chip FPGA, we could extend thread warping to handle any number of cores, and off-chip FPGAs, with minimal changes to the CAD tools and OS.

Instead of using thread warping, a designer could utilize standard static synthesis techniques (e.g., [11]) at compile time to create custom accelerators for threads, using specialized tools and/or languages. Although a static approach is an excellent technical solution, many software developers may resist such an approach due to the requirement of using a non-standard software tool flow. Thread warping hides the FPGA by dynamically synthesizing accelerators, allowing software developers to take advantage of the performance improvements of custom circuits without any changes to tool flow – just as multithreaded programs make use of more processors without rewriting or recompiling code. In addition, thread warping adapts the system to changing thread behavior, and different mixes of resident applications. For example, thread warping can potentially create different accelerator versions depending on the amount of available FPGA at different points during execution.

3. ON-CHIP CAD TOOLS

We define the following terms. A *thread creator* is a function that contains the API (application programming interface) call that creates a thread. A *thread function* is the function that a thread executes. A *thread* is the unit of execution that the OS schedules. A *thread group* is a collection of threads, created from the same instruction address, that share input data.

Figure 2(a) shows the on-chip CAD tool flow used by thread warping. Initially, *queue analysis* analyzes the thread queue to determine the union of waiting thread functions and a set of thread counts representing the occurrences of each thread function in the queue. Next, if accelerators do not already exist for the waiting thread functions, *accelerator synthesis* creates a custom accelerator circuit for each thread function and stores the accelerator in the *accelerator library*. When threads cannot be implemented entirely as FPGA circuits, accelerator synthesis may create an accelerator that acts as a coprocessor for one of the main microprocessors. To communicate between the microprocessor and accelerator, accelerator synthesis creates an updated software binary. It also creates a *thread group table* (TGT) that specifies all thread groups. After accelerator synthesis completes, *accelerator instantiation* determines the number of accelerators to place in the FPGA for each thread function. Accelerator instantiation outputs a circuit, which the *place & route* tool converts to an FPGA bitstream. Accelerator instantiation also creates a *schedulable resource list (SRL)* that informs the OS of available processing resources.

The OS invokes the on-chip CAD tools when the thread queue reaches a predefined size, and also periodically to adapt the system to changes. If new thread functions are present, then

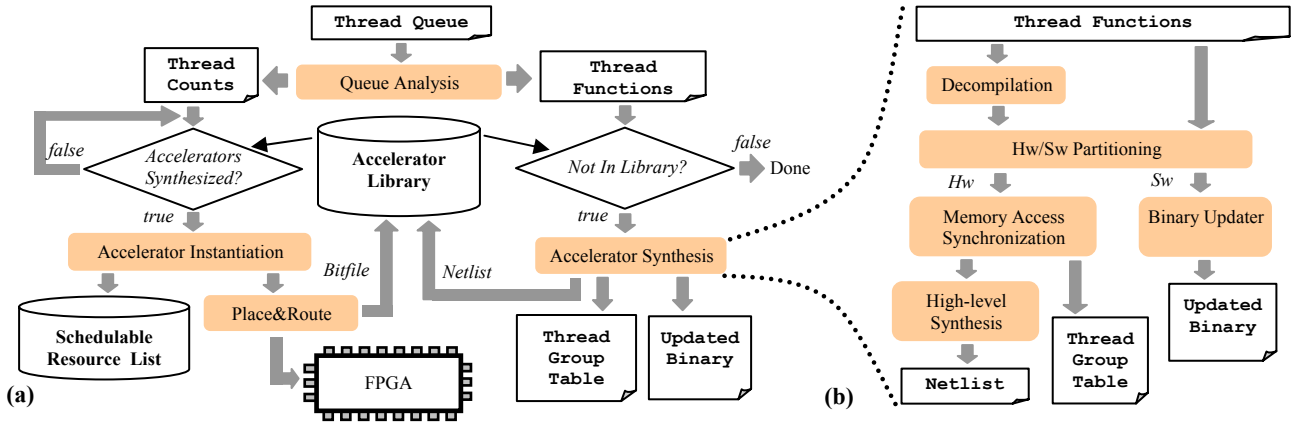

Figure 2: (a) *On-chip CAD* tool flow, which initially analyzes the thread queue and creates custom accelerators for waiting threads using the illustrated (b) *accelerator synthesis* tool flow.

accelerator synthesis considers creating new accelerators. Alternatively, if thread counts change, accelerator instantiation changes the types/amount of accelerators in the FPGA.

The current on-chip CAD tools support ARM11 binaries compiled with pthread functions calls. Pthread support is provided by the OS and is discussed in Section 4.

The CAD tools use numerous heuristics, but space does not allow detailed discussion of each. Key details regarding the novel techniques for accelerator synthesis and accelerator instantiation are discussed in the following sections.

3.1 Accelerator Synthesis

Figure 2(b) illustrates the tool flow of accelerator synthesis, which initially uses *decompilation* techniques [4], drawn from previous work in binary synthesis [27], to recover high-level constructs such as functions, function parameters, loops, if statements, arrays, etc. *Hw/Sw partitioning* analyzes decompiled thread functions, using previous techniques (e.g., [7]), to determine which regions to implement as circuits. Next, *memory access synchronization* analyzes thread functions, detects threads with similar memory access patterns, and combines those threads into thread groups that share memory channels and have synchronized execution. *High-level synthesis* converts the decompiled representation of the thread function (or other region) into a custom circuit, represented as a netlist. In addition, the *Binary updater* optionally modifies the original software binary to communicate with accelerators, in the case where an entire thread function is not implemented on the FPGA. We implement the binary updater using techniques from WARTS [12]. When implementing accelerators for entire thread functions, accelerator synthesis does not modify the thread binary.

3.1.1 Memory access synchronization

Inter-thread communication is a common bottleneck of multithreaded applications, and is often implemented using shared memory. This bottleneck poses a challenge for FPGA circuits, which may access shared memory using a DMA with a few channels. Thread warping further complicates memory accesses by requiring simultaneous access for possibly dozens of thread accelerators. Limited memory bandwidth restricts the number of DMA channels, forcing accelerators to multiplex data over a shared channel, reducing performance.

We observed that for some parallel applications, the required number of channels could be reduced because multiple threads may read the same data from memory. In this situation, memory access synchronization (MAS) can potentially combine memory accesses from multiple accelerators onto a single channel, and use a single read to service many accelerators. We refer to this situation as a "combinable" memory access.

To support combinable memory accesses, we developed MAS techniques to analyze memory access patterns of thread functions to detect overlapping accesses. MAS detects overlapping regions by determining all fixed-address memory reads in the control/data flow graph of each thread function. To increase the number of fixed-address reads, MAS unrolls loops to generate fixed-address reads for array calculations. MAS also treats stack-relative accesses as combinable for threads that have the same input parameters because in this case, the stack operations refer to the same data. MAS annotates combinable accesses, allowing high-level synthesis to combine data onto a single channel. MAS assigns all remaining non-fixed accesses onto a separate channel that multiplexes the data using different base addresses.

To implement a combined memory access, the OS synchronizes the execution of all involved accelerators. To achieve this synchronization, MAS analyzes thread creators and detects loops that create threads with combinable accesses, called thread groups. MAS determines the size of a thread group based on the number of threads created from a particular instruction address. Finally, MAS stores the thread group into the thread group table (TGT). The OS utilizes the TGT to synchronize all threads in the thread group, as discussed in Section 4.

3.1.2 High-level synthesis

To perform high-level synthesis, we use existing techniques for optimization, scheduling, resource allocation, and binding [24]. We also utilize smart buffers [10] to reduce memory accesses and improve pipeline throughput.

To support synthesis of thread synchronization, synthesis initially searches the control/data flow graph of each thread function to identify unsupported synchronization primitives. Our synthesis

tools currently support pthread create, join, mutex, and semaphore functions. If a thread function contains other calls, the OS will always schedule corresponding threads onto microprocessors. Software threads can utilize the entire pthread API.

For each semaphore, synthesis creates a controller that requests a semaphore operation from the OS and waits for the operation to complete. The semaphore-operation request triggers an interrupt, allowing the OS to service the request in an interrupt service routine. If an accelerator requests a lock for a locked semaphore, the OS adds the request to a synchronization queue, checked when another thread unlocks the semaphore. To handle synchronization requests from multiple accelerators, the OS uses an arbiter generated during accelerator instantiation. We considered hardware mutexes [2], but they would have required significant changes to the software binary.

Synthesis synchronizes thread groups by using a *memory access scheduler* circuit that initially time multiplexes all uncombined accesses over a single DMA channel and then triggers a combined access that delivers data to all involved accelerators.

The CAD tools use Xilinx ISE for register-transfer level synthesis. We currently execute ISE on a 3 GHz Pentium IV because no ARM11 version exists. However, our results are based on estimated ARM11 synthesis execution times, which we obtained from a comparison of runtimes on two similar systems in previous work [21]. In this paper, we focus mostly on recurring speedups, therefore the accuracy of the synthesis time estimates does not affect the results, and is intended to give an idea of how long the tools must execute before accelerators are available.

3.2 Accelerator Instantiation

After accelerator synthesis completes, accelerator instantiation determines the number of each type of accelerator to include in the FPGA for best performance. Some applications may benefit from using many accelerators for a single thread function, while others may benefit from a small number of accelerators for different thread functions. Furthermore, for some applications, the requirements may change during execution.

We map the problem to the 0-1 knapsack problem, where each item to be placed in the knapsack is an accelerator and the FPGA area represents the knapsack capacity. We map the area of each accelerator to the weight of each item. We define the profit of each accelerator as the product of the thread count percentage and the thread function speedup, defined as follows. Thread count percentage is the number of thread function occurrences in the queue divided by total queue size. Thread function speedup is the software performance of the thread divided by the accelerator performance. We then use a greedy knapsack heuristic with O(nlgn) complexity to generate a solution.

Accelerator instantiation also synthesizes an appropriate arbiter (Section 3.1.2) to handle the synchronization requests for each accelerator. Finally, accelerator instantiation passes the netlist for all accelerators to placement and routing to generate a bitfile for the FPGA. In addition, accelerator instantiation updates the schedulable resource list with the new resource amounts.

3.3 Placement and Routing

We use Xilinx ISE, running on a Pentium IV, for placement and routing, and again estimate ARM11 execution times as discussed in Section 3.1.2. Alternatively, because existing commercial tools were not designed for fast dynamic synthesis, we could use specialized placement and routing techniques designed for fast FPGA mapping [20] or just-in-time (JIT) FPGA compilation [22], which reduce placement and routing times by 10x to 46x. Specialized FPGAs fabrics have also been introduced to enable fast placement and routing [1].

After placement and routing, the CAD tools store the bitfile in the accelerator library, allowing all future executions to immediately utilize the accelerators.

3.4 Limitations

The original code must use the pthread API. Ideally, the approach would support any thread API by detecting use of threads in the binary during execution. However, such detection is an open problem. Alternatively, the technique could use instruction sets, such as Java byte code, that have explicit thread information.

The thread functions synthesized by the CAD tools must only use create, join, mutex, and semaphore primitives. We are currently extending the CAD tools to support other types of synchronization, in addition to other communication techniques such as message passing.

Accelerator instantiation is currently targeted toward applications using a boss-worker thread model. For applications with many threads performing different tasks, improved strategies based on historical profiles may be needed, and remains as future work.

The decompilation techniques are also a potential limitation. If decompilation does not recover enough high-level information, the resulting hardware may be inefficient. However, previous work has shown that decompilation often makes synthesis from a software binary competitive with high-level synthesis [23][27].

4. OPERATING SYSTEM SUPPORT

To support thread synchronization, the OS API includes all pthread functions, which are also used by commercial operating systems such as VxWorks [28]. We plan to also consider other APIs, such as the task functions from VxWorks.

The OS scheduler handles the scheduling of threads onto both microprocessors and a set of custom thread accelerators specified by the schedulable resource list (SRL) shown in Figure 2.

The scheduler maintains a thread queue that stores threads waiting for processing resources. For the thread at the queue head, the scheduler checks the SRL to determine what resources are available and compatible with the thread. The scheduler gives priority to the fastest resource that is compatible with the thread function, which is usually an accelerator.

A problem occurs when there are no accelerators for the first thread in the queue, and no microprocessors are available. Scheduling algorithms often examine only the head of the queue to achieve O(1) complexity. For thread warping, there may exist other threads in the queue that have available accelerators. However, if the thread at the queue head cannot be scheduled, then the remaining threads in the queue also cannot be scheduled. To avoid this problem, the scheduler scans the thread queue until finding a thread that can be scheduled. This more complex scheduler has O(n) complexity, where n is the maximum size of the thread queue. However, the scheduler often avoids the worst

case by not scanning the queue if no resources are available, or if available resources do not apply to any waiting threads.

The scheduler is non-preemptive for threads executing on accelerators, and preemptive for software threads. By utilizing non-preemptive scheduling, the scheduler avoids having to save FPGA state – a task far more difficult than for software [17]. The scheduler is invoked anytime a thread is created/completed, anytime a lock is released, or anytime a software thread is blocked by a synchronization request.

The scheduler also synchronizes the execution of thread groups in the TGT, as in Section 3.1.1. Each entry in the TGT consists of a thread group queue and a maximum size. When an application creates a thread from a thread group, the scheduler adds that thread to the appropriate queue in the TGT. When a TGT queue reaches the maximum specified size, the scheduler moves the entire queue onto the main thread queue, and schedules the entire thread group simultaneously. If the number of accelerators for the corresponding thread function is less than the size of the thread group, the scheduler moves the queue from the TGT when the size of the queue matches the number of accelerators.

5. EXPERIMENTS

5.1 Experimental Setup

To evaluate the performance of the framework, we developed a C++ simulator consisting of approximately 30,000 lines of code, including the on-chip CAD tools (excluding Xilinx ISE) and the OS simulation.

The simulator creates a *parallel execution graph* (PEG) that represents thread-level parallelism. Each node of the PEG is a *sequential execution block* (SEB) – a block that either ends with a pthread call or represents the end of a thread. PEG edges represent synchronization between SEBs. For a SEB to be schedulable, all SEB parents must have finished. Each SEB also specifies synchronization, such as mutex locks, that must succeed before the SEB can complete. To generate the PEG, the simulator uses pthread wrapper functions with knowledge of parent and children SEB blocks. Using those functions, the simulator creates a PEG from a single sequential execution of the application.

For each SEB, the simulation determines software and hardware performance, and synthesis execution time. The simulator determines software performance using SimpleScalar [3] and hardware performance using VHDL simulation of the synthesized accelerator. In the case that SEB performance is data dependent, one can manually annotate the performance.

After determining performances of each SEB, the simulator uses the PEG to perform an event-driven simulation of the architecture. The simulator schedules SEBs to processing resources, while updating the state of the architecture, OS, and FPGA as the CAD tools execute. The simulation completes after scheduling all SEBs.

The simulator currently does not simulate arbitration overhead for multi-core microprocessor memory accesses, and instead assumes all cores can simultaneously access memory. Such an assumption results in optimistic software execution times, therefore the reported circuit speedups are *pessimistic*.

The simulator maintains memory coherency by not scheduling software threads while accelerators are executing, and by flushing each cache upon accelerator completion. This restriction simplifies simulation by not requiring simulation of a coherency protocol. Because this restriction artificially limits parallelism, the reported circuit speedups are again *pessimistic.*

5.2 Comparison of Multi-Core Systems and Thread Warping

We compare multicore architectures consisting of ARM11 microprocessors running at 400 MHz, with an architecture performing thread warping, consisting of four microprocessors and a Xilinx Virtex IV XC4VLX15 FPGA, which has the area equivalent to approximately 36 ARM11 cores. The FPGA runs at the frequency determined by placement and routing for each set of accelerators, ranging from 100 to 300 MHz.

To evaluate the framework, we developed multithreaded versions of image processing benchmarks. *Fir* is a finite impulse response filter. *Prewitt* performs Prewitt edge detection. *Linear* performs a linear search. *Moravec* performs the Moravec algorithm. *Wavelet* performs a wavelet transform. *Maxfilter* outputs the maximum of a window of pixels. *3DTrans* performs 3-dimensional graphic transformations. We also use an *N-body* simulation to test longer-running scientific-computing applications. While an ARM11 is not typically used for scientific computing, the framework can easily be extended to high-performance processors.

Figure 3 shows application speedups for multiprocessor systems and for thread warping (*TW*) compared to the performance of a 4-core system. On average, thread warping resulted in a speedup of 130x compared to the 4-core architecture. The geometric mean was 38x. Thread warping was 11x faster than the 64-core system, outperforming the 64-core system for all examples except *Fir, Linear,* and *3Dtrans*. These results for thread warping represent the performance *after* creating custom accelerators and do not represent the initial execution of the application. The initial execution of the applications would have almost identical performance as the 4-core system, except for *N-body*, which would still have been 8x faster because the original execution

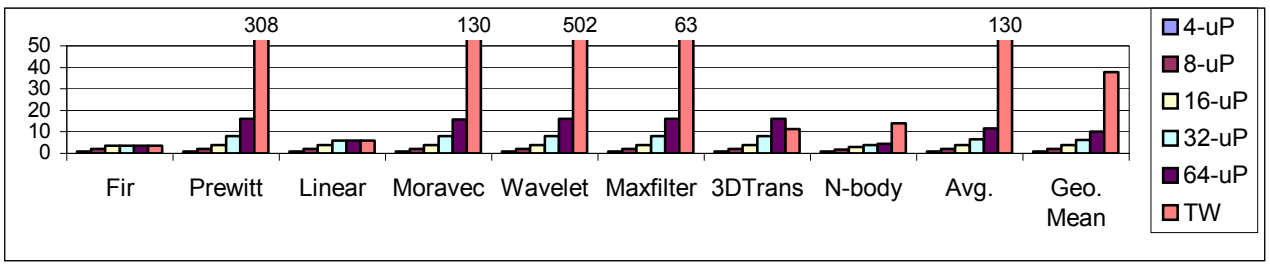

Figure 3: Speedups of multithreaded applications on multiprocessor systems versus thread warping (TW). The speedups are compared to the execution time of the 4-microprocessor (uP) system. On average, TW achieved a 130x speedup.

time was 9.4 hours. The synthesis execution times ranged from 22 minutes to 48 minutes, averaging 32 minutes.

Thread warping achieved large speedups for *Prewitt*, *Moravec*, *Wavelet*, and *Maxfilter* because those examples have no dependencies between threads, allowing for most of the application to be parallelized. Of course, examples exist that are not amenable to FPGA implementations, such as many desktop applications, for which thread warping would not currently achieve speedup. However, for those applications that are amenable to FPGA implementations, the improvements are significant.

6. CONCLUSIONS

We introduced a framework for dynamic synthesis of thread accelerators, or *thread warping*, which transparently creates custom FPGA circuits for threads. To enable the technique, we introduced CAD methods for accelerator synthesis and accelerator instantiation, and integrated those methods into an architecture that allows the OS to schedule threads onto both microprocessors and accelerators. Using an extensive architecture simulator, the approach resulted in overall application speedups ranging from 4x for an FIR filter to 502x for a wavelet transform.

Although some of the performance gains could be obtained using multiprocessing platforms with more microprocessors instead of an FPGA, thread warping achieves speedups across a wider range of applications by parallelizing applications at finer-granularities than just the thread level. Furthermore, thread warping utilizes FPGAs, which may be lower cost than multiprocessing platforms due to FPGAs' widespread usage in a variety of systems yielding economy of scale.

7. ACKNOWLEDGMENTS

This work was funded in part by the National Science Foundation (CNS-0614957) and the Semiconductor Research Corporation (2005-HJ-1331).

8. REFERENCES

[1] Amerson, R., Carter, R., Culbertson, W., Kuekes, P., Snider, G., and Albertson, L. Plasma: an FPGA for million gate systems. In *Proceedings of Int. Symp. on Field Programmable Gate Arrays (FPGA)*, 1996, 10-16.

[2] Andrews, D., Niehaus, D., and Ashenden, P. Programming models for hybrid CPU/FPGA chips. *IEEE Computer*, 37, 1 (2004), 118-120.

[3] Burger, D. and Austin, T. The simplescalar tool set, version 2.0. *SIGARCH Computer Architecture News*, 25, 3 (1997), 13-35.

[4] Cifuentes, C. *Reverse Compilation Techniques*. PhD Thesis, Queensland University of Technology, 1994.

[5] Cray XD1. http://www.cray.com/products/xd1, 2005.

[6] Dellson, A., Sandberg, G., and Möhl, S. *Turning FPGAs into Supercomputers*. Cray User Group, 2006.

[7] Eles, P., Peng, Z., Kuchchinski, K., and Doboli, A. System level hardware/software partitioning based on simulated annealing and tabu search. *Journal on Design Automation for Embedded Systems (DAES)*, Springer, 2, 1 (1997), 5-32.

[8] Fin, A., Fummi, F., and Signoretto, M. SystemC: a homogenous environment to test embedded systems. In *Proceedings of Int. Workshop on Hardware/Software Codesign (CODES)*, 2001, 17-22.

[9] Grimpe, E. and Oppenheimer, F. Extending the SystemC synthesis subset by object oriented features. In *Proceedings of Int. Conf. on Hardware/Software Codesign and System Synthesis (CODES/ISSS)*, 2003, 25-30.

[10] Guo, Z., Buyukkurt, A.B., and Najjar, W. Input data reuse in compiling window operations onto reconfigurable hardware. In *Proceedings of Symposium on Languages, Compilers and Tools for Embedded Systems (LCTES)*, 2004, 249-256.

[11] Gupta, S., Dutt, N., Gupta, R., and Nicolau, A. SPARK : a high-level synthesis framework for applying parallelizing compiler transformations. In *Proceedings of Int. Conf. on VLSI Design*, 2003.

[12] Hill, M., Larus, J., Lebeck, A., Talluri, M., and Wood, D. Wisconsin architectural research tool set. *SIGARCH Computer Architecture News*. 21, 4 (1993).

[13] IBM. The Cell Architecture. http://domino.research.ibm.com, 2006.

[14] Schleupen, K., Lekuch, S., Mannion, R., Guo, Z., Najjar, W., and Vahid, F. Dynamic partial FPGA reconfiguration in a prototype microprocessor system. In *Proceedings of Int. Conf. on Field Programmable Logic And Applications*, 2007.

[15] Intel Quad-Core Xeon. http://www.intel.com , 2007.

[16] Jung, H. and Ha, S. Hardware synthesis from coarse-grained dataflow specification for fast hw/sw cosynthesis. In *Proceedings of Int. Conf. on Hardware/Software Codesign and System Synthesis (CODES/ISSS)*, 2004, 24-29.

[17] Koch, D., Haubelt, C., and Teich, J. Efficient hardware checkpointing: concepts, overhead analysis, and implementation. In *Proceedings of Int. Symp. on Field Programmable Gate Arrays (FPGA)*, 2007, 188-196.

[18] M. LaPedus. *Intel Tips Teraflops Programmable Processor*. EE Times, September 2006.

[19] Lu, J., Chen, H., Yew, P., and Hsu, W. Design and implementation of a lightweight dynamic optimization system. *Journal of Instruction-Level Parallelism*, 6 (Jun 2004), 1-24.

[20] Ludwig, S. *Fast Hardware Synthesis Tools and a Reconfigurable Coprocessor*. Ph.D. Thesis, ETH Zurich, 2005.

[21] Lysecky, R., Stitt, G., and Vahid, F. Warp processors. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 11, 3 (2006), 659-681.

[22] Lysecky, R., Vahid, F., and Tan, S. A study of the scalability of on-chip routing for just-in-time FPGA compilation. In *Proceedings of IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM)*, 2005, 57-62.

[23] Mittal, G., Zaretsky, D., Tang, X., and Banerjee, P. Automatic translation of software binaries onto FPGAs. In *Proceedings of ACM Design Automation Conference (DAC)*, 2004, 389-394.

[24] De Micheli, G. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1994.

[25] Rakhmatov, D. and Vrudhula, S. Hardware-software bipartitioning for dynamically reconfigurable systems. In *Proceedings of Int. Workshop on Hardware/Software Co-Design (CODES)*, 2002, 145-150.

[26] SGI Altix. http://www.sgi.com/products/servers/altix/

[27] Stitt, G. and Vahid, F. New decompilation techniques for binary-level co-processor generation. In *Proceedings of IEEE/ACM Int. Conf. on Computer-Aided Design (ICCAD),* 2005, 547-554.

[28] VxWorks RTOS. http://www.windriver.com/vxworks/, 2007.

[29] Xilinx Virtex II Pro, http://www.xilinx.com, 2006.

[30] Xilinx Virtex IV, http://www.xilinx.com, 2006.

[31] Zhang, W., Calder, B., and Tullsen, D. An event-driven multithreaded dynamic optimization framework. In *Proceedings of Int. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, 2005, 87-98.