Warp Processors

ROMAN LYSECKY
University of Arizona
GREG STITT
University of California, Riverside
and
FRANK VAHID
University of California, Riverside

We describe a new processing architecture, known as a warp processor, that utilizes a field-programmable gate array (FPGA) to improve the speed and energy consumption of a software binary executing on a microprocessor. Unlike previous approaches that also improve software using an FPGA but do so using a special compiler, a warp processor achieves these improvements completely transparently and operates from a standard binary. A warp processor dynamically detects the binary's critical regions, reimplements those regions as a custom hardware circuit in the FPGA, and replaces the software region by a call to the new hardware implementation of that region. While not all benchmarks can be improved using warp processing, many can, and the improvements are dramatically better than those achievable by more traditional architecture improvements. The hardest part of warp processing is that of dynamically reimplementing code regions on an FPGA, requiring partitioning, decompilation, synthesis, placement, and routing tools, all having to execute with minimal computation time and data memory so as to coexist on chip with the main processor. We describe the results of developing our warp processor. We developed a custom FPGA fabric specifically designed to enable lean place and route tools, and we developed extremely fast and efficient versions of partitioning, decompilation, synthesis, technology mapping, placement, and routing. Warp processors achieve overall application speedups of 6.3X with energy savings of 66% across a set of embedded benchmark applications. We further show that our tools utilize acceptably small amounts of computation and memory which are far less than traditional tools. Our work illustrates the feasibility and potential of warp processing, and we can foresee the possibility of warp processing becoming a feature in a variety of computing domains, including desktop, server, and embedded applications.

F. Vahid is also with the Center for Embedded Computer Systems at the University of California at Irvine.

This research was supported by the National Science Foundation (CCR-0203829) and the Semiconductor Research Corporation (2003-HJ-1046G).

Authors' addresses: R. Lysecky, Department of Electrical and Computer Engineering, University of Arizona, Tucson, AZ, 85721; email: rlysecky@ece.arizona.edu; G. Stitt, F. Vahid, Department of Computer Science and Engineering, University of California, 900 University Ave., Riverside, CA 92521.

Categories and Subject Descriptors: C.1.3 [**Processor Architectures**]: Other Architecture Styles—*Adaptable architectures*; C.3 [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems—*Real-time and embedded systems*

General Terms: Design, Experimentation, Performance

Additional Key Words and Phrases: Warp processors, hardware/software partitioning, FPGA, configurable logic, just-in-time (JIT) compilation, dynamic optimization, hardware/software codesign

1. INTRODUCTION

Extensive research over the past two decades has demonstrated the benefits that often can be obtained by reimplementing a software application's critical regions, or critical kernels, as a custom circuit coprocessor on a field-programmable gate array (FPGA). While many products today feature such coprocessors on FPGAs alongside a microprocessor [Altera 2006; Christensen 2004; D. H. Brown Associates 2004; Morris 2005; Xilinx 2004a; Xilinx 2005a], and while recent commercial compilers offer to automatically partition critical software regions to FPGAs or ASICs [Critical Blue 2005] or to create custom coprocessors tightly integrated within the processor itself [Tensilica 2006], the significantly different and costlier tool flows associated with FPGA coprocessors have prevented the use of such coprocessors in mainstream software flows.

We define a software application's critical region as a loop or subroutine that is frequently executed, accounting for perhaps 10% or more of total application execution time. For many applications, the majority of execution time may come from just a few critical regions. Therefore, while speeding up a critical region accounting for only 10% of the execution time may not provide extremely high overall speedups in itself, by speeding up several such critical regions, the overall performance improvement can be significant. Implementing a critical region on an FPGA, commonly called hardware/software partitioning, can sometimes result in high speedups of 100X–1000X or more for that region. Of course, the impact of this speedup on the overall application speedup depends on the contribution percentage of the region to overall application execution time, per Amdahl's law. Furthermore, not all applications have regions amenable to speedup with an FPGA. Nevertheless, for many applications, researchers and commercial vendors have observed overall application speedups of 10X–100X [Balboni et al. 1996; Berkeley Design Technology 2004; Chen et al. 2004; Eles et al. 1997; Ernst et al. 1993; Gajski et al. 1998; Guo et al. 2005; Henkel and Ernst 1997; Keane et al. 2004; Stitt and Vahid 2005; Stitt et al. 2005], and in some cases close to 1000X [Böhm et al. 2002; Venkataramani et al. 2001], obtained by implementing critical regions on an FPGA.

The advent of single-chip microprocessor/FPGA devices makes hardware/software partitioning even more attractive. Such devices include one or more microprocessors and an FPGA fabric on a single chip, and typically include efficient mechanisms for communication between the microprocessor and FPGA, along with shared memory. Such devices first appeared in the late 1990s from Atmel [2005] and Triscend [Matsumoto 2000; Triscend 2003] with low-end microprocessors and FPGAs supporting tens of thousands of gates. Altera [2005] developed Excalibur devices incorporating an ARM9 processor and a million-gate FPGA fabric. Xilinx offers Virtex-II Pro [2004b] and Virtex-4 FX [2005b] devices incorporating one or more PowerPC processors with an FPGA fabric having tens of millions of gates. These devices all implement the processor as a hard core, not as a circuit mapped onto the FPGA fabric itself. Given the appearance of FPGAs supporting tens of millions of gates, and soon hundreds of millions of gates, and also knowing that microprocessors may require only tens or hundreds of thousands of gates, we can see that any FPGA can implement microprocessors as soft cores mapped on the FPGA fabric itself.

Electronically programming bits onto an FPGA is fundamentally the same as programming a microprocessor. Like a microprocessor, an FPGA is an off-the-shelf part. We can program an FPGA by downloading a bitstream into the FPGA's memory, just as we program a microprocessor. Thus, conceptually, a compiler can partition an application into a microprocessor part and an FPGA coprocessor part, and indeed, such compilers do exist, although mostly in the research domain [Balboni et al. 1996; Böhm et al. 2002; Critical Blue 2005; Eles et al. 1997; Gajski et al. 1998; Gokhale and Stone 1998; Hauser and Wawrzynek 1997; Henkel and Ernst 1997; Stitt and Vahid 2002; Xilinx 2000a]. Unfortunately, such compilers require a significant departure from traditional software tool flows. First, the compiler must determine the critical regions of a software application, and such determination typically requires profiling. Profiling, while conceptually straightforward, is often not a standard part of compilation, especially in embedded systems where executing an application often involves complicated time-dependent interactions with the application's environment and makes setting up simulations difficult. Second, the compiler must generate a binary for the microprocessor and a binary for the FPGA coprocessor, and the latter is by no means standard. Thus, partitioning compilers lose the important concept of a standard binary and the associated benefits of portability.

Recently, researchers showed that by using decompilation techniques, designers could perform desktop hardware/software partitioning starting from binaries rather than from high-level code, with resulting competitive performance and energy [Banerjee et al. 2004; Mittal et al. 2004; Stitt and Vahid 2002, 2005]. Binary-level partitioning opens the door to dynamic hardware/software partitioning, in which an executing binary is dynamically and transparently optimized by moving software kernels to on chip configurable logic, a process we call *warp processing*. Warp processors, originally proposed in [Stitt et al. 2003], provide designers with the ability to program using a high-level language, such as C, while exploiting the underlying FPGA to improve performance and reduce energy consumption with no required knowledge of the FPGA. Such transparent optimizations can be quite beneficial, as programming in a high-level language is about five times more productive than programming in VHDL [Vissers 2004].

2. FPGA COPROCESSING

Implementing a critical region as a circuit on an FPGA may yield high performance speedups compared to implementing on a microprocessor when the

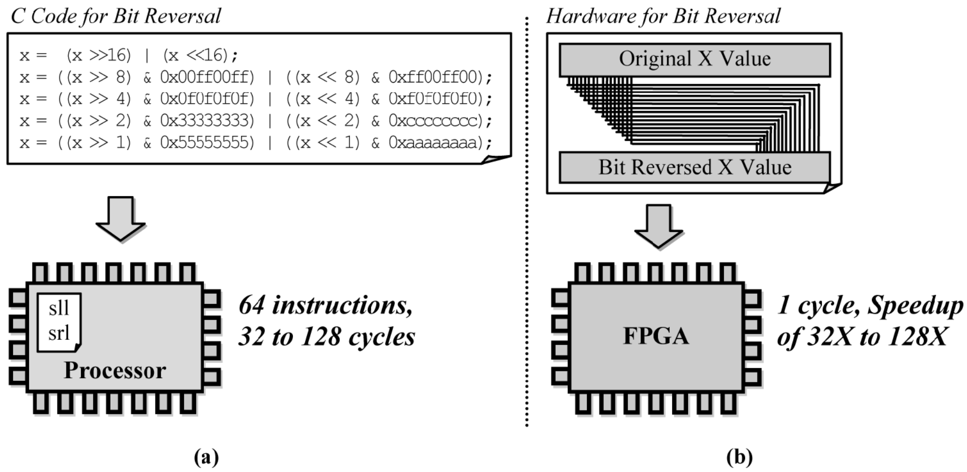

Fig. 1. A comparison of a bit reversal in (a) software on a microprocessor and (b) hardware on an FPGA.

critical region involves extensive bit-level manipulation or its code is highly parallelizable.

Bit-level operations are much less efficient on a microprocessor because each bit-level operation requires a separate instruction or several instructions. An example of the efficiency of FPGA bit-level manipulations is a bit reversal operation, shown in Figure 1. Figure 1 (a) presents an efficient software implementation of a bit reversal that requires approximately 64 instructions to reverse one 32-bit integer [Press et al. 1992]. Depending on the microprocessor, these 64 instructions could require anywhere from 32 to 128 cycles to complete. However, as shown in Figure 1 (b), a bit reversal implemented as a hardware circuit in an FPGA requires only wires to compute and can be performed in a single cycle, achieving a speedup ranging from 32X to 128X (assuming equal cycle lengths).

Furthermore, an FPGA can generally implement parallelizable code much more efficiently than even a VLIW (very large instruction word) or multiple-issue microprocessor. Whereas a microprocessor might be able to execute several operations in parallel, an FPGA can potentially implement thousands of operations in parallel. The finite impulse response (FIR) filter shown in Figure 2 is an example of highly parallelizable code that can potentially greatly benefit from implementing the code on an FPGA. Figure 2 (a) shows the execution of the FIR filter on a microprocessor requiring many multiply-accumulate operations and executing for thousands of cycles. Alternatively, Figure 2 (b) shows an FPGA implementation of the FIR filter in which the hardware circuit within the FPGA performs multiplications in parallel and implements the accumulation as a tree of adders (assuming sufficient FPGA resources). By parallelizing multiply-accumulate operations, the FPGA implementation can achieve a speedup of at least 100X.

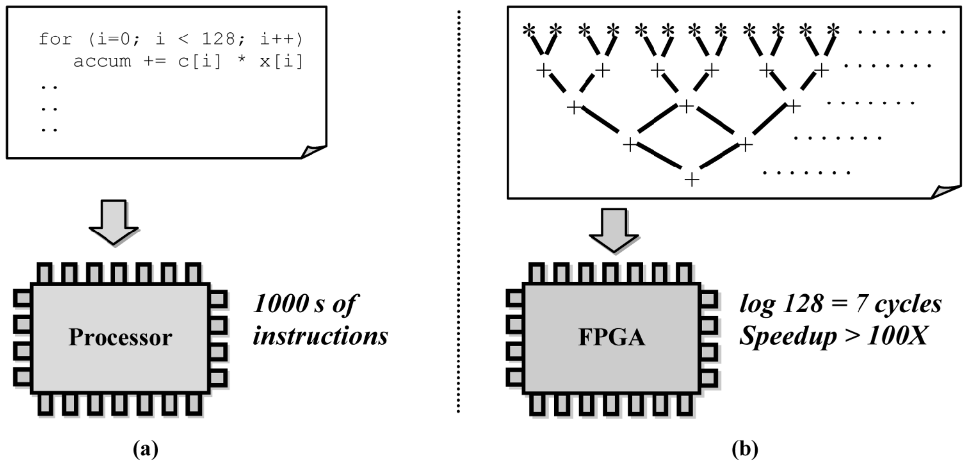

Fig. 2. FIR filter implemented in (a) software on a microprocessor and (b) hardware on an FPGA.

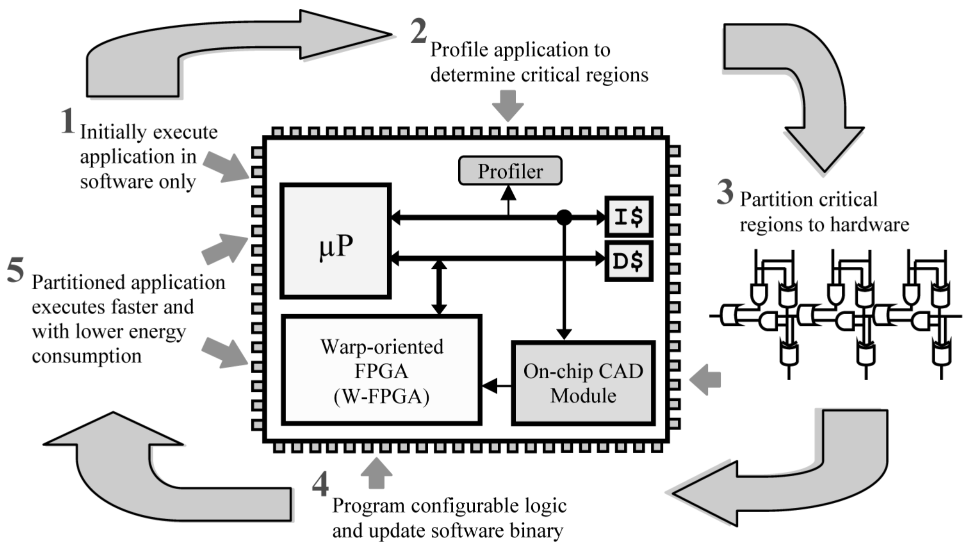

Fig. 3. Warp processor architecture/overview.

3. COMPONENTS OF A WARP PROCESSOR

Figure 3 provides an overview of a warp processor, highlighting the steps performed during dynamic hardware/software partitioning. A warp processor consists of a main processor with instruction and data caches, an efficient on chip profiler, our warp-oriented FPGA (W-FPGA), and an on-chip computer-aided design module. Initially, a software application executing on a warp processor will execute only on the main processor. During the execution of the application, the profiler monitors the execution behavior of the application to determine the critical kernels within. After identifying the critical regions, the on-chip CAD module executes the Riverside on-chip CAD tools (ROCCAD) to reimplement the critical software regions as a custom hardware component within the W-FPGA.

We include a *profiler* within each warp processor to determine the critical kernels within the executing application that the warp processor could implement as hardware. Dynamic profiling is a widely studied problem for which many solutions exist [Lysecky et al. 2004a; Zagha et al. 1996; Zhang et al. 1997; Zilles and Sohi 2001]. Typical profilers instrument code, thus changing program behavior and requiring extra tools. Instead, we incorporate a nonintrusive profiler that monitors the instruction addresses seen on the instruction memory bus [Gordon-Ross and Vahid 2003]. Whenever a backward branch occurs, the profiler updates a cache of 16 8-bit entries that store the branch frequencies. When any of the registers storing the branch frequencies becomes saturated, the profiler shifts all 16 registers to the right by one bit, thereby maintaining a list of relative branch frequencies while ensuring all branch frequency registers do not eventually become saturated. The profiler uses roughly 2000 gates along with a small cache of only a few dozen entries, with a small associativity, to save area and power. Furthermore, through simulations of our warp processor design, we have found that the profiler can accurately determine the critical regions of an application within ten branch frequency register saturations. Using this methodology, the profiler is able to properly select the correct critical kernels for partitioning, for all the applications we considered. Further details of the profiler design and accuracy can be found in Gordon-Ross and Vahid [2003].

After profiling the application to determine the critical regions, the on-chip CAD module executes our partitioning, synthesis, mapping, and routing algorithms. ROCCAD first analyzes the profiling results for the executing application and determines which critical region the warp processor should implement in hardware. After selecting the region, ROCCAD decompiles the critical region into a control/dataflow graph and synthesizes the critical kernel to produce an optimized hardware circuit that is then mapped onto our W-FPGA through technology mapping, placement, and routing. Finally, ROCCAD configures the configurable logic and updates the executing application's binary code to utilize the hardware within the configurable logic fabric. During the binary update, the warp processor must ensure that the main processor is not currently executing within the critical region.

Currently, we implement the on-chip CAD module as a separate ARM7 processor including both caches and separate instruction and data memories, which can either be located on chip or off-chip, depending on what is acceptable for any given warp processor implementation. Alternatively, we could eliminate the need for the on-chip CAD module by executing our on-chip CAD tools as a software task on the main processor sharing computation and memory resources with the main application. We can also envision a multiprocessor system in which we incorporate multiple warp processors on a single device. In such a system, there is no need to additionally incorporate multiple on chip CAD modules, as a single on-chip CAD module is sufficient for supporting each of the processors in a round robin or similar fashion. Furthermore, we can again implement the CAD tools as a software task executing on any of the multiple processors.

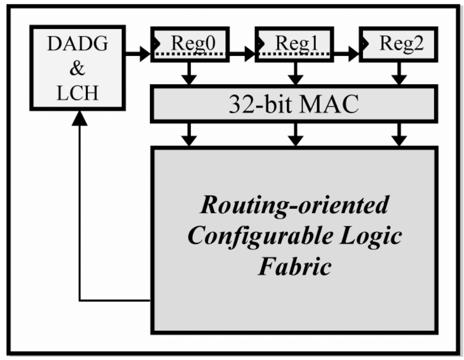

Fig. 4. A warp-oriented field programmable gate array (W-FPGA).

Finally, after reimplementing an application's critical kernels as hardware, programming the W-FPGA with the hardware configuration, and updating the application's binary to interface with the hardware, the application executes on the warp processor using a mutually exclusive execution model whereby either the main processor or the W-FPGA is active at any given time. Using this implementation, the main processor and the W-FPGA can access the same data cache, thereby avoiding any cache coherency and/or consistency issues. Furthermore, we examined the potential benefits of allowing parallel execution of the processor and configurable logic fabric and found that parallel execution did not yield significant performance improvements for most applications.

Simulating our warp processor design, we find that our warp processor provides excellent performance and energy benefits for the embedded applications we analyze in this article. Warp processing is ideally suited for embedded systems that repeatedly execute the same application, or set of applications, for extended periods, and especially for systems in which software updates and backwards compatibility are essential. As such, a warp processor can quickly determine which critical regions to implement in hardware and continue utilizing the same hardware configuration, either for the duration of the product's lifetime or until the software is updated. However, warp processing can also be incorporated into other domains such as desktop computing, high performance servers, personal digital assistants, etc. Within these domains, warp processing could prove to be extremely useful for data-intensive applications such as image or video processing, data analysis, scientific research, or even games, as these applications typically execute for an extended period of time during which the warp processor could partition the critical kernels to hardware. Additionally, short-running applications that execute many times can also benefit from warp processing, as long as the warp processor can remember the application's hardware configuration.

4. WARP-ORIENTED FPGA

Figure 4 shows the overall organization of our W-FPGA, consisting of a data address generator (DADG) with loop control hardware (LCH), three input and output registers, a 32-bit multiplier-accumulator (MAC), and our routing-oriented configurable logic fabric (RCLF) presented in Lysecky and Vahid [2004]. The W-FPGA handles all memory accesses to and from the configurable logic using the data address generator. Furthermore, the data retrieved and stored to and from each array is located within one of the three registers Reg0, Reg1, and Reg2. These three registers also act as inputs to the configurable logic fabric and can be mapped as inputs to the 32-bit multiplier-accumulator or directly mapped to the configurable logic fabric. As a final step, we connect the outputs from the configurable logic fabric as inputs to the three registers using a dedicated bus.

Since we are targeting critical loops that usually iterate many times before completion, the W-FPGA must be able to access memory and to control the execution of the loop. We include a data address generator with loop control hardware in our FPGA design to handle all memory accesses, as well as to control the execution of the loop. The data address generator within the W-FPGA can handle memory accesses that follow regular access patterns. Such data address generators and loop control hardware are often incorporated into digital signal processors to achieve zero loop overhead, meaning that cycles are not wasted in computing loop bounds and sequential memory addresses. Loop control hardware is typically capable of executing a loop for a specific number of iterations. While we can determine the loop bounds for many critical loops, loops can also contain control code within that terminates the loop's execution. For example, in a $C/C{+}{+}$ implementation to perform a lookup in an array, once we have found the desired value, we will typically terminate the loop's execution using a *break* statement. Therefore, the loop control hardware within the W-FPGA will control the loop's iterations, assuming a predetermined number of iterations, but will allow for terminating the loop's execution using an output from the configurable logic fabric.

In the applications we analyzed for developing warp processors, we frequently found common operations within an application's critical regions, including addition, subtraction, and multiplication. Furthermore, while we often see multiplications in the critical code regions, they are often in the form of a multiply-accumulate operation. Implementing a multiplier with a small configurable logic fabric is generally slow and requires a large amount of logic and routing resources. Therefore, we include a dedicated multiplier-accumulator within the W-FPGA to conserve resources and provide fast performance.

Figure 5 (a) shows our routing-oriented configurable logic fabric presented in Lysecky and Vahid [2004]. Our RCLF consists of an array of configurable logic blocks (CLBs) surrounded by switch matrices for routing between CLBs. Each CLB is connected to a single switch matrix to which all inputs and outputs of the CLB are connected. We handle routing between CLBs using the switch matrices, which can route signals in one of four directions to an adjacent SM (represented as solid lines in the figure), or to an SM two rows apart vertically or two columns apart horizontally (represented as dashed lines).

Figure 5 (b) shows our combinational logic block architecture. Each CLB consists of two 3-input 2-output LUTs, which provide the equivalent of a CLB consisting of four 3-input single output LUTs with fixed internal routing. We chose 3-input 2-output LUTs to simplify our technology mapping and placement

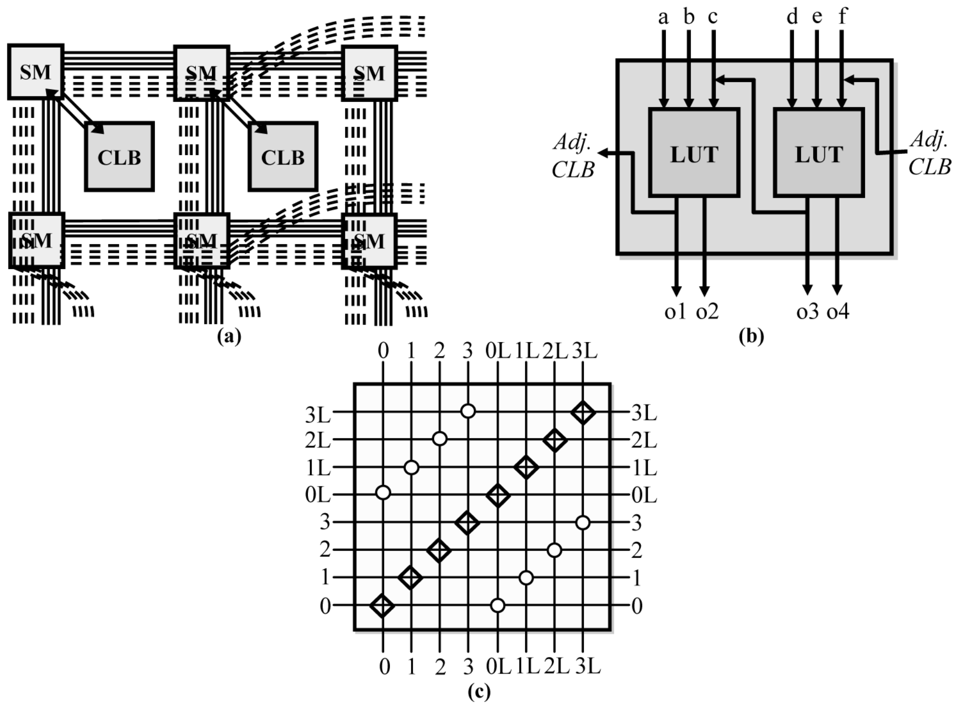

Fig. 5. (a) A routing-oriented configurable logic fabric, (b) combinational logic block, and (c) switch matrix architecture.

algorithms by restricting the choices our tools will analyze to determine the final circuit configuration. Additionally, the CLBs are capable of supporting carry chains through direct connections between horizontally adjacent CLBs and within the CLBs through internal connections between adjacent LUTs. Hardware components, such as adders, comparators, etc., frequently require carry logic, and so providing support for carry chains simplifies the required routing for many hardware circuits.

The size of our LUTs and CLBs is very important, as it directly impacts both area resources and delays within our configurable logic fabric, as well as the complexity of the tools needed to map a circuit to the configurable logic fabric. Several studies have analyzed the impacts of LUT size on both area and timing [Chow et al. 1999; Singh et al. 1992]. These studies have shown that lookup tables with five or six inputs result in circuits with the best performance, while LUTs with three or four inputs are still reasonable. Another study analyzed the impacts on cluster size, the number of single output LUTs within a CLB, and on the speed and area of various circuits [Marquardt et al. 2000]. Their findings indicate that cluster sizes of 3 to 20 LUTs were feasible, and a cluster size of 8 produced the best tradeoff between area and the delay of the final circuits. However, while we would like to incorporate large cluster sizes within our configurable logic fabric, such clusters allow more flexibility during technology mapping and placement phases which in turn require more complex technology mapping and placement algorithms to handle the added intricacy.

Finally, Figure 5 (c) shows our switch matrix architecture. Each switch matrix is connected using eight channels on each side of the switch matrix; four short channels routing between adjacent nodes and four long channels routing between every other switch matrix. Routing through the switch matrix can only connect a wire from one side with a given channel to another wire on the same channel, but on a different side of the switch matrix. Additionally, each of the four short channels is paired with a long channel and can be connected together within the switch matrix (indicated as a circle where two channels intersect), allowing wires to be routed using short and long connections. In other words, the switch matrix can route a wire on a single channel between two different sides of the switch matrix, connecting short, long, or short and long channels. Designing the switch matrix in this manner simplifies the routing algorithm by only allowing the router to route a wire using a single pair of channels throughout the configurable logic fabric.

Commercially available FPGAs consist of similar routing resources, but typically are capable of routing between switch matrices that are much further apart and often include routing channels spanning an entire row or column. While such flexible routing resources are beneficial in terms of creating compact designs with less routing overhead, they require complex place and route tools not amenable to on-chip execution. Therefore, we chose to limit the complexity of routing resources to allow for simplified place and route algorithms. In doing so, we were able to develop a set of fast, lean place and route tools that can execute 10X faster using 18X less memory than existing desktop-based place and route tools [Lysecky et al. 2005]. While our configurable logic design is fundamentally similar to existing FPGAs and can achieve comparable performance to existing commercial FPGAs for many circuits, its limited routing resources result in lower clock frequencies for large circuits that require a large percentage of the routing resources. However, for partitioning SW kernels, the inclusion of the data address generator and multiply-accumulator within our W-FPGA reduces the complexity and size of the circuit implemented on the configurable logic, thereby allowing for faster clock frequencies and alleviating any potential degradation in performance. The interested reader can find further details on the configurable logic fabric design in Lysecky and Vahid [2004].

5. RIVERSIDE ON-CHIP COMPUTER-AIDED DESIGN TOOLS

Warp processors require the development of lean versions of partitioning, decompilation, behavioral, register-transfer (RT), and logic synthesis, technology mapping, placement, and routing algorithms. However, the traditional desktop-based counterparts of these tools typically require long execution times often ranging from minutes to hours, large amounts of memory resources often exceeding 50 megabytes, and large code sizes possibly requiring hundreds of thousands of lines of source code. However, the on chip CAD algorithms and tools incorporated within warp processors must have very fast execution times while using small instruction and data memory resources. Therefore, we developed Riverside on-chip CAD (ROCCAD) tools, designed specifically to provide very

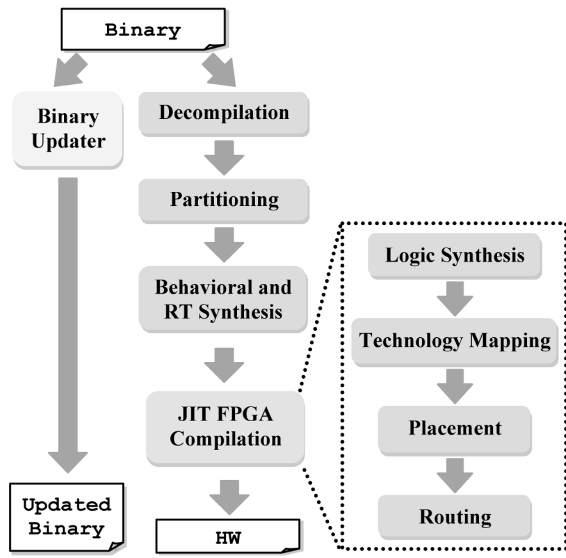

Fig. 6. ROCCAD on-chip CAD tool chain.

fast execution times while minimizing the data amount of memory used during execution and providing excellent results.

Figure 6 presents the tool chain flow for our on chip CAD tools, ROCCAD, executed on the on-chip CAD module's processor. Starting with the software binary, *decompilation* converts the software loops into a high-level representation more suitable for synthesis. Much of the decompilation techniques we utilize are based on Cifuente et al.'s work [1998, 1999] in binary translation. Decompilation first converts each assembly instruction into equivalent register transfers which provides an instruction set independent representation of the binary. Once decompilation converts the instructions into register transfers, the decompilation tool builds a control flow graph for the software region and then constructs a data flow graph by parsing the semantic strings for each register transfer. The parser builds trees for each register transfer and then combines the trees into a full data flow graph through definition-use and use-definition analysis. After creating the control and data flow graphs, the decompiler applies standard compiler optimizations to remove the overhead introduced by the assembly code and instruction set. After recovering a control/data flow graph, the decompilation process analyzes it to recover high-level constructs, such as loops and *if* statements [Cifuentes 1996].

ROCCAD then performs *partitioning* by analyzing the critical software kernels, determined by the on-chip profiler, to evaluate which software kernels are most suitable for implementing in hardware. Using a simple partitioning heuristic, the partitioning algorithm determines which critical kernels the warp processor will implement within the configurable logic architecture to maximize speedup while reducing energy. Next, ROCCAD uses behavioral and register-transfer (RT) synthesis to convert the control/data flow graph for each critical kernel into a hardware circuit description. The RT synthesis then converts the hardware circuit descriptions into a netlist format, which specifies the hardware circuits using Boolean expressions for every output of the circuit.

Finally, the on-chip CAD tools execute just-in-time (JIT) FPGA compilation tools to map the hardware binary onto the underlying configurable logic fabric. The JIT FPGA compiler first executes *logic synthesis* to optimize the hardware circuit. Starting with the Boolean equations representing the hardware circuit, the JIT FPGA compiler initially creates a directed acyclic graph of the hardware circuit's Boolean logic network. The internal nodes of the graph correspond to any simple 2-input logic gate such as AND, OR, XOR, NAND, etc. We then perform logic synthesis to optimize the logic network using the Riverside on-chip minimizer (ROCM), a simple two-level logic minimizer presented in Lysecky and Vahid [2003]. Starting with the input nodes, we traverse the logic network in a breadth-first manner and apply logic minimization at each node. ROCM's logic minimization algorithm uses a single *expand* phase to achieve good optimization. While a more robust two-level logic minimizer could achieve better optimization for larger examples, the simplified algorithm is better suited for on-chip execution of warp processors.

After logic synthesis, the JIT FPGA compiler performs *technology mapping* to map the hardware circuit onto the CLBs and LUTs of the configurable logic fabric. The technology mapper uses a greedy hierarchical graph-clustering algorithm. It first performs a breadth-first traversal of the input-directed acyclic graph starting with the output nodes, and combines nodes to create LUT nodes corresponding 3-input single output LUTs. Once we identify the single output LUT nodes, the technology mapper again performs a breadth-first traversal starting from the output nodes and combines nodes wherever possible to form the final 3-input 2-output LUTs, which are a direct mapping to the underlying configurable logic fabric. Finally, the technology mapper again traverses the graph, now representing the technology-mapped hardware circuit, and packs the LUTs together into CLBs by identifying situations in which we can utilize the routing resources between adjacent LUTs, such as when the output from one LUT is an input to another LUT.

After mapping the hardware circuit into a network of CLBs, the JIT FPGA compiler places the CLB nodes onto the configurable logic. The placement algorithm is a greedy dependency-based positional algorithm that first determines a placement of the CLB nodes within the hardware circuit relative to each other. The placement algorithm starts by determining the critical path within the circuit and places these nodes into a single horizontal row within the RCLF. It then analyzes the remaining nonplaced nodes to determine the dependency between them and the nodes already placed. Based upon these dependencies, for each unplaced node, we place the node either above (input to an already placed node) or below (uses an output from an already placed node) and as close as possible to the dependent node. During this placement procedure, the algorithm also attempts to utilize the routing resources between adjacent CLBs within the RCLF, whenever possible. After determining the relative placement of LUT nodes, the placement algorithm superimposes and aligns the relative placement onto the configurable logic fabric, making minor adjustments at the edges if needed.

We then perform *routing* between inputs, outputs, and CLBs with the configurable logic fabric using the Riverside on-chip router (ROCR) presented in Lysecky et al. [2004b] and Lysecky et al. [2005]. ROCR utilizes the general approach of Versatile Place and Route's (VPR's) routability-driven router, allowing both overuse of routing resources and illegal routes, and eliminates illegal routing through repeated routing iterations [Betz et al. 1999; Betz and Rose 1997]. ROCR starts by initializing the routing costs within a routing resource graph representing the configurable logic fabric. For all unrouted nets, ROCR uses a greedy routing approach to route the net. During the greedy routing process, for each sink within the net, ROCR determines a route between either the unrouted sink and net's source or the nearest routed sink. At each step, it restricts the router to only those paths within a bounding box of the current sink and the chosen location to which they are routing. After all nets are routed, if illegal routes exist as the result of overusing routing channels, then ROCR rips up only the illegal routes and adjusts the routing costs of the entire routing resource graph. ROCR uses the same routing cost model of VPR's routability-driven router. However, in addition, ROCR incorporates an adjustment cost. During the process of ripping up illegal routes, ROCR adds a small routing adjustment cost to all routing resources used by an illegal route. During the routing process, an early routing decision can force the routing algorithm to choose a congested path. Hence, the routing adjustment cost discourages the greedy routing algorithm from selecting the same initial route and enables the algorithm to attempt a different routing path in subsequent iterations. Once we determine a valid global routing, ROCR performs detailed routing in which we assign the channels used for each route. Detailed routing starts by constructing a routing conflict graph. Two routes conflict when both pass through a given switch matrix and assigning them the same channel would result in an illegal routing within it. ROCR assigns the routing channels by determining a vertex coloring of the routing conflict graph. While many approaches for vertex coloring exist, we chose to use Brelaz's [1979] vertex coloring algorithms. Brelaz's algorithm is a simple and greedy algorithm, producing good results while not increasing ROCR's overall memory consumption. For those routes that ROCR cannot assign to a legal channel, ROCR rips up the illegal routes, adjusts the routing costs of all nodes along the illegal route (as described before), and reroutes the illegal routes. It finishes routing a circuit when a valid routing path and channel assignment has been determined for all nets.

Finally, the *binary updater* handles updating the software binary to utilize the hardware for loops. We replace the original software instructions for the loop with a jump to hardware initialization code. The initialization code first enables the hardware by writing to a memory-mapped register or port that is connected to the hardware enable signal. Following the enable instruction, we generate the code responsible for shutting the microprocessor down into a power-down sleep mode. After finishing execution, the hardware then asserts a completion signal that causes a software interrupt. This interrupt wakes up the microprocessor, which resumes normal execution. Finally, we add a jump instruction to the end of the hardware initialization code that jumps to the end of the original software loop.

Table I. Processor and FPGA Operating Frequencies (of commercially available single-chip microprocessor/FPGA devices)

Device	Proc. Freq. (MHz)	FPGA Freq. (MHz)	Proc.:FPGA Freq.
Xilinx Virtex-4 FX (0.09 μm)	450	500	1:1.1
Xilinx Virtex-II Pro (0.13 μm)	400	320	1:0.8
Altera Excalibur (0.18 μm)	200	180	1:0.9
Atmel FPSLIC (0.18 μm)	40	33	1:0.8
Triscend A7 (0.18 μm)	60	60	1:1.0
Average:			**1:0.92**

6. EXPERIMENTAL RESULTS

Considering a warp processor system with a dedicated on-chip CAD module, we must be able to execute the on-chip tools on a small, embedded microprocessor, such as an ARM7, using limited instruction and data memory resources while providing fast execution times. Furthermore, on a system in which the CAD tools execute as a task on the one of the main processors, the algorithms must still use limited memory resources and execute quickly so as not to impact the execution of the main application. ROCCAD tools require 34,720 lines of C code corresponding to a binary size of 327 kilobytes. Additionally, the ROCCAD algorithms execute for an average of only 1.2 seconds on a 40 MHz ARM7 processor, requiring a maximum of 3.6 megabytes of data memory during execution for the benchmarks described in the following.

We compare our warp processors with a traditional hardware/software partitioning approach targeting an FPGA, comparing speedup and energy reduction of critical regions for 15 embedded systems benchmarks from NetBench [Memik et al. 2001], MediaBench [Lee et al. 1997], EEMBC [EEMBC 2005], Powerstone [Malik et al. 2000], as well as our on-chip logic minimization tool ROCM [Lysecky and Vahid 2003]. Instead of restricting our analysis to a specific processor and FPGA, we compare the performance and energy savings of both approaches in a device independent manner by analyzing software and hardware implementations with respect to the ratio of execution frequency and power consumption of the processor to that of the configurable logic. Thus, in utilizing ratios to characterize both the processor and FPGAs operating frequency and power consumption, we can analyze the benefits of warp processing independently of the specific technology process we are utilizing. This is due to the fact that the ratios between processor and FPGA operating frequency are likely to remain the same with successive technology generations.

In determining the processor to FPGA operating frequency and power consumption ratios, we analyzed the published performance and power consumption data for several commercially available single-chip microprocessor/FPGA devices. Table I provides a summary of the maximum operating frequencies of several single-chip microprocessor/FPGA devices. The Xilinx Virtex-4 FX is the fastest of the devices, integrating two or more 450 MHz PowerPC processors within a configurable logic fabric operating at a maximum frequency of 500 MHz [Xilinx 2005b]. Xilinx also offers the second fastest device, the Virtex-II Pro, integrating two or more 400 MHz PowerPC processors within a configurable logic fabric operating at a maximum frequency of 320 MHz [Xilinx 2004b]. Altera's

Table II. Operating Frequency and Power Consumption Ratios (between a Low power ARM7 processor, a traditional FPGA, and the W-FPGA)

Device	Proc.:FPGA Freq.	Proc.:FPGA Power
FPGA	1:0.92	1:3.0
W-FPGA	1:1.0	1:2.25

Excalibur combines a 200 MHz ARM9 processor with an FPGA operating at a maximum frequency of 180 MHz [Altera 2005]. Atmel's Field Programmable System Level Integrated Circuit (FPSLIC) combines a 40 MHz AVR processor with an FPGA capable of executing at 33 MHz [Atmel 2005]. Finally, Triscend offered the A7 that combines a processor and an FPGA both executing at a maximum frequency of 60 MHz [Triscend 2003]. Table I further provides the ratio of processor frequency to FPGA frequency for these platforms. In the best case, the Virtex-4 FX device achieves a ratio of 1:1.1, whereas the Xilinx Virtex-II Pro and Atmel FPSLIC both have a ratio of 1:0.8. On average, these single-chip devices have a processor to FPGA frequency ratio of 1:0.92. Furthermore, as processors continue to increase in speed with each subsequent process technology node, the operating frequencies of FPGAs will also increase while the ratio of processor frequency to FPGA frequency will remain relatively constant, as exhibited by the successive offering from Xilinx [2000b].

Table II presents the operating frequency and power consumption ratios between a low power ARM7 processor and a traditional FPGA, and between a low power ARM7 processor and our W-FPGA. The processor to FPGA operating frequency ratio is based on the average presented in Table I. We calculated the processor to FPGA power consumption ratio by evaluating the power consumption of the critical kernels for several embedded benchmark applications, implemented as software, executing an on a low power processor and hardware executing on an FPGA. For each of the 15 NetBench, MediaBench, EEMBC, and Powerstone benchmarks considered, we implemented the applications' critical regions as hardware by manually designing a VHDL implementation and synthesizing the design for a Xilinx Virtex-E FPGA using Xilinx ISE 4.1 [Xilinx 2006]. Using the Xilinx Virtex Power Estimator along with information provided by Xilinx ISE, we determined the power consumed by the FPGA for each critical region. We then calculated the processor to FPGA power consumption ratio by comparing the average power consumed by the Virtex-E FPGA to the power consumption of a low power ARM7 processor, both of which are implemented using 0.18 μm technology. On average, the FPGA consumes three times as much power as the ARM7 processor for the critical regions of the applications we consider, corresponding to a ratio of 1:3.0. Furthermore, as the spreadsheet-based Xilinx Virtex Power Estimator is not as accurate as the XPower tool, we conducted a similar analysis using several of the embedded benchmark applications with almost identical results.

The simplicity of the W-FPGA's configurable logic fabric allows us to achieve higher execution frequencies and lower power consumption compared to a traditional FPGA (implemented using the same process technology) for the embedded applications considered. We analyzed both the execution frequency and

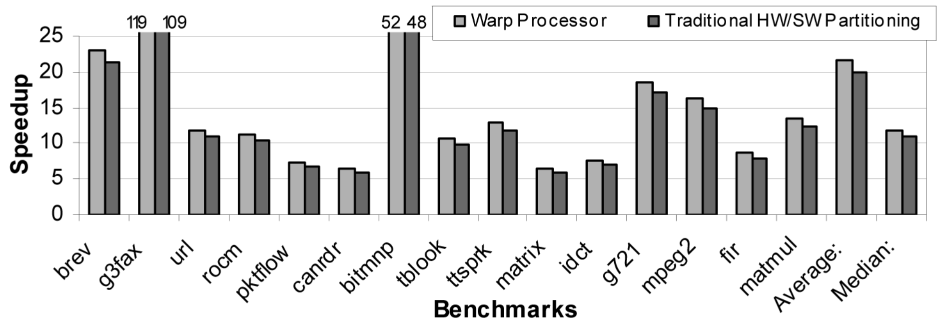

Fig. 7. Critical region speedup of a single critical region implemented using warp processors and traditional hardware/software partitioning, targeting an FPGA for NetBench, MediaBench, EEMBC, and Powerstone benchmark applications.

the power consumption of our W-FPGA compared to an existing FPGA. To determine the performance and power consumption of our W-FPGA, we implemented our configurable logic architecture in VHDL and synthesized the design using the Synopsys Design Compiler targeting the UMC 0.18 μm technology library. Using the synthesized fabric along with gate-level simulations, we determined the delay and power consumption of individual components within our configurable logic architecture. We note that this ASIC implementation of our configurable logic is less efficient than a full custom layout would be in terms of performance, power, and area, and as such, our estimates for the performance and power consumption of the W-FPGA are slightly pessimistic. We implemented the critical regions for all benchmark applications using our on-chip CAD tools and determined the maximum execution frequency by determining the critical path within each placed and routed design, and further calculated the power consumption for each circuit. In addition to considering the delay and power consumption of the configurable logic and switch matrices, we also considered the delay and power consumption associated with the short and long wire segments used for routing the hardware design. Finally, we compared the execution frequency of our W-FPGA with the Xilinx Virtex-E FPGA, as the Virtex-E FPGA uses a 0.18 μm process. On average, our W-FPGA can achieve clock frequencies 1.5x faster than the Xilinx FPGA and consumes 25% less power. Therefore, our warp processors should exhibit a ratio of processor frequency to W-FPGA frequency of 1:1 and a power consumption ratio of 1:2.25.

Figures 7 and 8 highlight the critical region speedup and critical region energy reduction for the *single most* critical kernel using warp processors and traditional hardware/software partitioning targeting an FPGA for all 15 benchmarks using the operating and power consumption ratios summarized in Table II. In calculating the speedups of the two approaches, we determined all software execution cycles using the SimpleScalar simulator [Burger and Austin 1997], and hardware execution cycles using gate-level simulations of the partitioned critical regions and our W-FPGA. For the traditional hardware/software partitioning, we manually designed the hardware implementations of the critical regions in VHDL to determine the number of cycles required for the hardware execution, considering the same critical

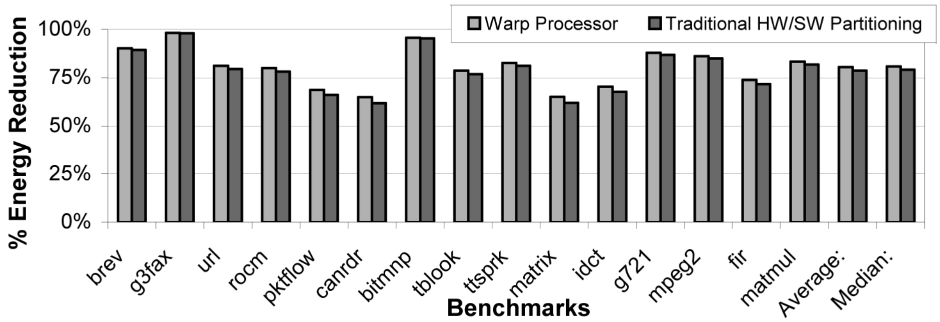

Fig. 8. Critical region energy reduction of a single critical region implemented using warp processors and traditional hardware/software partitioning, targeting an FPGA for NetBench, MediaBench, EEMBC, and Powerstone benchmark applications.

$$
\begin{aligned}
E_{total} &= E_{Proc} + E_{HW} \\
E_{Proc} &= P_{Proc\ (active\ )} \times t_{active} + P_{Proc\ (idle\ )} \times t_{idle} \\
E_{HW} &= P_{HW} \times t_{active} + P_{static} \times t_{total}
\end{aligned}
$$

Fig. 9. Equations for determining energy consumption after hardware/software partitioning.

regions partitioned by warp processing the traditional hardware/software partitioning.

We calculated the energy required for the critical regions after partitioning using the equations in Figure 9. The total energy consumption, E_{total}, is the sum of the energy consumed by the processor, E_{Proc}, and the energy consumed by the hardware configuration, E_{HW}. The energy consumed by the processor consists of that during the initialization of the hardware configuration (computed as the processor's active power consumption, $P_{Proc(active)}$, multiplied by the initialization time, t_{init}), and the energy consumed while the critical region is executing in hardware, computed as the processor's idle power consumption, $P_{Proc(idle)}$, multiplied by the active time for the hardware configuration, t_{active}. The processor power consumption is based on the power consumption of the ARM7 executing at its maximum operating frequency, 100 MHz. The energy consumed by the hardware configuration consists of the energy consumed during execution of the hardware (computed as the power consumption of the hardware configuration, P_{HW}, multiplied by the active time for the hardware configuration, t_{active}), and also the static energy consumed by configurable logic during the entire execution time of the application (computed as the static power consumed by the configuration logic, P_{static}, multiplied by the total execution time, t_{total}).

On average, our warp processor achieves a critical region speedup of 22X with an average energy reduction of 80%. In comparison, the traditional hardware/software partitioning approach targeting an FPGA achieves a critical region speedup of 20X and an energy reduction of 79%. Additionally, warp processors achieve a critical region speedup of over 100X for the benchmark *g3fax*. For eight of the applications, including *brev*, *g3fax*, *url*, *bitmnp, ttsprk, g721, mpeg2,* and *matmul*, the warp processor provides an energy reduction of over 80% for the applications' single most critical regions. The increased performance of the

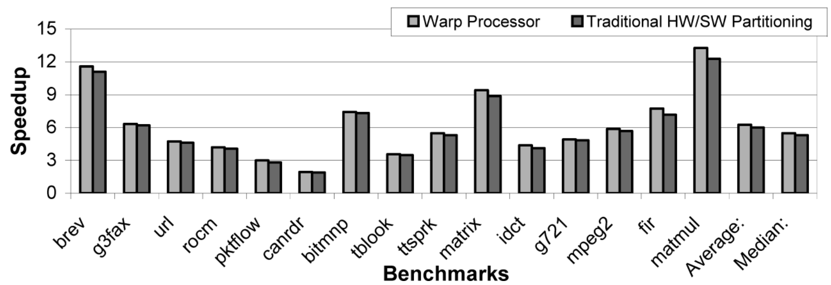

Fig. 10. Overall application speedups of NetBench, MediaBench, EEMBC, and Powerstone benchmark applications implemented using warp processors (supporting up to four critical regions) and traditional hardware/software partitioning targeting an FPGA.

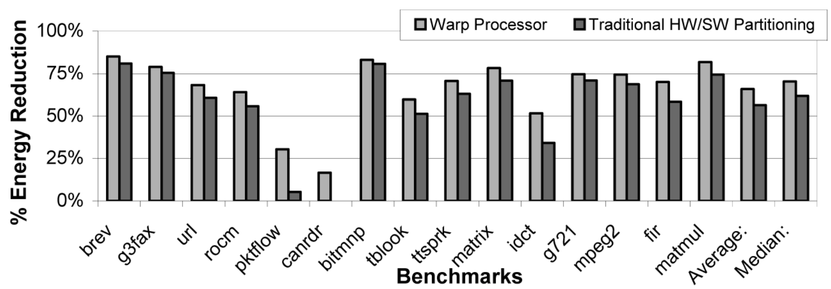

Fig. 11. Overall energy reduction of NetBench, MediaBench, EEMBC, and Powerstone benchmark applications implemented using warp processors (supporting up to four critical regions) and traditional hardware/software partitioning targeting an FPGA.

warp processor compared to traditional hardware/software partitioning can be partially attributed to the inclusion of customized hardware resources within the W-FPGA design and to the simplicity of the W-FPGA's configurable logic fabric.

Figures 10 and 11 present *overall application* speedup and energy reduction using warp processors (supporting up to four critical regions) and traditional hardware/software partitioning targeting an FPGA for all 15 benchmarks using the operating and power consumption ratios summarized in Table II. On average, warp processors provide an overall application speedup of 6.3X while reducing energy consumption by 66%. Alternatively, the traditional hardware/software partitioning targeting an FPGA results in an average overall application speedup of 6X and an energy reduction of 56%. For two applications, *brev* and *matmul*, warp processors achieve a speedup of greater than 10X while reducing energy consumption by more than 80%.

We analyzed the energy consumption of warp processors compared to several different processor alternatives, ranging from processors designed for low power, to voltage-scalable processors, to high-performance processors. Figure 12 presents the average critical region energy consumption of all 15 benchmark applications for 10 different processor configurations, normalized to the energy consumption of the XScale processor executing at the maximum possible frequency of 502 MHz. While executing the benchmarks' critical kernels on an Intel Pentium 4 processor may provide extremely fast execution, the power

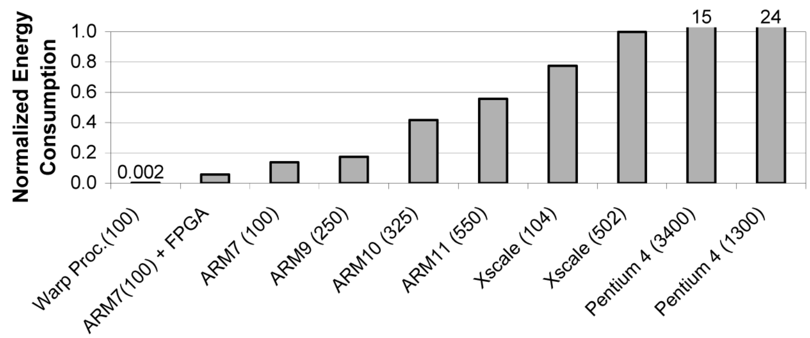

Fig. 12. Average overall energy consumption of NetBench, MediaBench, EEMBC, and Powerstone benchmark applications using various processors normalized to the XScale processor at 502 MHz *(processor frequency reported in megahertz)*.

demands of the Pentium processor lead to extremely high energy consumption, requiring roughly 15X to 24X more energy than the XScale processor. While high-performance processors are typically not designed for low power consumption, voltage-scalable processors are designed for flexibility in performance and power consumption by allowing a designer to adjust both the operating voltage and the frequency. By reducing the operating voltage and frequency of the XScale processor, we can execute it at 102 MHz while reducing energy consumption by 22%. Alternatively, many processors are designed to provide low power consumption, such as the ARM family of processors, and can often be found in embedded systems where battery lifetime is a primary concern. Although the 100 MHz ARM7 is the slowest processor, it provides the lowest energy consumption of all the traditional processors we evaluated. However, a warp processor incorporating a 100 MHz ARM7 processor and our W-FPGA achieves the lowest overall energy consumption, requiring less than 5% of the energy consumed by the XScale processor and approximately one-fifth of one percent of the energy consumed by the Pentium 4 processor.

Finally, we evaluated the area required to implement our W-FPGA. As described earlier, we synthesized the VHDL implementation of our W-FPGA using a Synopsys Design Compiler targeting the UMC 0.18 μm technology library. Our configurable logic architecture, including the address generators, multiply-accumulator, and supporting logic, has an area of 14.2 mm^2, roughly corresponding to 852,000 gates. Compared to a low power processor utilized for the main processor with our warp processor design, the W-FPGA requires 3X more area than an ARM7 processor with an 8 kilobyte cache. However, our configurable logic architecture only requires 1.2X more area than an ARM9 with 32 kilobytes of cache, a reasonable choice for the main processor. Thus, when combined with the main processor, a warp processor achieves an average speedup of 6.3X, using between 2.2X and 4X more area than a small low power processor alone. As an alternative comparison, our W-FPGA is approximately equivalent in area to a 64 kilobyte cache.

Additionally, for each benchmark application we determined the amount of resources required to implement its critical kernels within the W-FPGA's

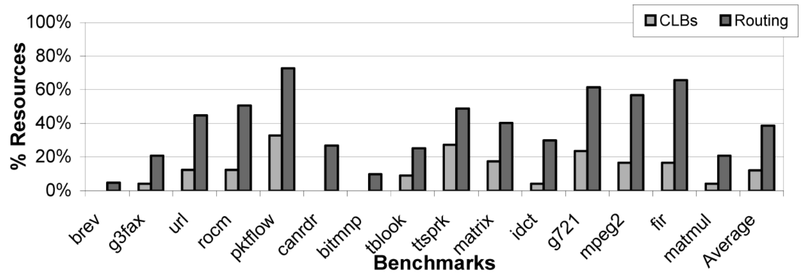

Fig. 13. Percentage of RCLF configurable logic blocks (CLBs) and routing resources used to implement critical regions of NetBench, MediaBench, EEMBC, and Powerstone benchmark applications.

routing-oriented configurable logic fabric. Figure 13 presents the percentage of configurable logic blocks and routing resources within our RCLF required to implement the critical regions of the NetBench, MediaBench, EEMBC, and Powerstone benchmark applications. On average, the applications' critical kernels required 12% of the available CLBs and 39% of the routing resources. The benchmark *pktflow* required the largest amount of resources, utilizing 33% of the CLBs and 73% of the routing resources to implement its critical kernels. At the other extreme, *brev* required the fewest resources within the RCLF, only utilizing 5% of the routing resources.

7. CONCLUSIONS AND FUTURE WORK

Our work demonstrates that the basic concept of warp processing, namely the concept of dynamically mapping software kernels to an on-chip FPGA for performance and energy improvements, is possible. Although the work described in this article is extensive, involving several years of development and showing reasonable speedups of 6.3X and energy savings of 66% (and up to 10X and 80% in some examples) for several standard benchmarks, extensive future work is still required. Better speedups and energy savings may be obtained through improved lean partitioning, synthesis, placement, and routing, through advanced decompilation methods that can detect higher-level constructs, and through the use of embedded multipliers, block RAMs, and other components. Another hurdle to overcome is the memory bottleneck, which limits speedups in many benchmarks. Advanced memory access methods may reduce this bottleneck, and we are investigating this direction [Stitt et al. 2005] of advancement. Improving the warp-oriented FPGA represents another direction for further progress, we are presently fabricating a prototype of the W-FPGA through collaboration with Intel as an effort in this direction. Furthermore, studies of desktop-based applications are needed to determine the extent to which warp processing can improve such applications, which may involve pointers, extensive access to data structures in memory, and execution spread over more regions than is typical in embedded applications. For both embedded and desktop applications, we also need to address the impact and interplay of an operating system executing within our warp processor architecture. As the operating system is an essential component in scheduling software execution and managing hardware resources, the operating system needs to be aware of warp processing and potentially control the warp processor's dynamic partitioning. Additionally, research must be done to compare the warp precessor's use of silicon area to other alternatives that could also speedup software binaries. Yet, the demonstrated impressive 100X-1000X speedups of existing hardware/software partitioning methods mean that the potential benefits of warp processing are quite significant.

REFERENCES

ALTERA CORP. 2006. Customer showcase. http://www.altera.com/corporate/cust_successes/customer_showcase/view_product/csh-vproduct-nios.jsp.

ALTERA CORP. 2005. Excalibur embedded processor solutions. http://www.altera.com/products/devices/excalibur/exc-index.html.

ATMEL CORP. 2005. FPSLIC (AVR with FPGA), http://www.atmel.com/products/FPSLIC/.

BALBONI, A., FORNACIARI W., AND SCIUTO, D. 1996. Partitioning and exploration in the TOSCA co-design flow. In *Proceedings of the International Workshop on Hardware/Software Codesign (CODES)*, 62–69.

BANERJEE, P., MITTAL, G., ZARETSKY, D., AND TANG, X. 2004. BINACHIP-FPGA: A tool to map DSP software binaries and assembly programs onto FPGAs. In *Proceedings of the Embedded Signal Processing Conference (GSPx)*.

BERKELEY DESIGN TECHNOLOGY, INC. 2004. http://www.bdti.com/articles/info_eet0207fpga.htm#DSPEnhanced%20FPGAs.

BETZ, V., ROSE, J., AND MARQUARDT, A. 1999. *Architecture and CAD for Deep-Submicron FPGAs*. Kluwer Academic, Hingham, Mass.

BETZ, V. AND ROSE, J. 1997. VPR: A new packing, placement, and routing for FPGA research. In *Proceedings of the International Workshop on Field Programmable Logic and Applications (FPLA)*, 213–222.

BÖHM, W., HAMMES, J., DRAPER, B., CHAWATHE, M., ROSS, C., RINKER, R., AND NAJJAR, W. 2002. Mapping a single assignment programming language to reconfigurable systems. *J. Supercomput. 21*, 117–130.

BRELAZ, D. 1979. New methods to color the vertices of a graph. *Commun. ACM 22*, 251–256.

BURGER, D. AND AUSTIN, T. 1997. The SimpleScalar tool set, version 2.0. *SIGARCH Comput. Architecture News, 25*, 3.

CHEN, W., KOSMAS, P., LEESER, M., AND RAPPAPORT, C.. 2004. An FPGA implementation of the two-dimensional finite-difference time-domain (FDTD) algorithm. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA)*, 97–105.

CHRISTENSEN, F. 2004. A scalable software-defined radio development system. *Xcell J.*, Winter.

CHOW, P., SEO, S., ROSE, J., CHUNG, K., PAEZ-MONZON, G., AND RAHARDJA, I. 1999. The design of an SRAM-based field-programmable gate array, part I: Architecture. *IEEE Trans. Very Large Scale Integration Syst. (TVLSI), 7*, 2, 191–197.

CIFUENTES, C. 1996. Structuring decompiled graphs. In *Proceedings of the International Conference on Compiler Construction*. Lecture Notes in Computer Science, vol. 1060, 91–105.

CIFUENTES, C., SIMON, D., AND FRABOULET, A. 1998. Assembly to high-level language translation. Department of Computer Science and Electrical Engineering, University of Queensland. Tech. Rep. 439.

CIFUENTES, C., VAN EMMERIK, M., UNG, D., SIMON, D., AND WADDINGTON, T. 1999. Preliminary experiences with the use of the UQBT binary translation framework. In *Proceedings of the Workshop on Binary Translation*, 12–22.

CRITICAL BLUE. 2005. http://www.criticalblue.com.

D.H. BROWN ASSOCIATES. 2004. Cray XD1 brings high-bandwidth supercomputing to the mid-market. White Paper prepared for Cray, Inc., http://www.cray.com/downloads/dhbrown_crayxd1_oct2004.pdf.

EEMBC. 2005. The Embedded Microprocessor Benchmark Consortium. http://www.eembc.org.

ELES, P., PENG, Z., KUCHCHINSKI, K., AND DOBOLI, A. 1997. System level hardware/software partitioning based on simulated annealing and Tabu search. *Kluwer's Design Automation for Embedded Systems 2*, 1, 5–32.

ERNST, R., HENKEL, J., AND BENNER, T. 1993. Hardware-software cosynthesis for microcontrollers. *IEEE Des. Test Comput. 10*, 4, 64–75.

GAJSKI, D., VAHID, F., NARAYAN, S., AND GONG, J. 1998. SpecSyn: An environment supporting the specify-explore-refine paradigm for hardware/software system design. *IEEE Trans. Very Large Scale Integration Syst. (TVLSI) 6*, 1, 84–100.

GOKHALE, M. AND STONE, J. 1998. NAPA C: Compiling for hybrid RISC/FPGA architectures. In *Proceedings of the Symposium on FPGAs for Custom Computing Machines (FCCM)*, 126.

GORDON-ROSS, A. AND VAHID, F. 2003. Frequent loop detection using efficient non-intrusive on-chip hardware. In *Proceedings of the Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, 117–124.

GUO, Z., BUYUKKURT, B., NAJJAR, W., AND VISSERS, K. 2005. Optimized generation of data-path from C codes. In *Proceedings of the Design Automation and Test in Europe Conference (DATE)*, 112–117.

HAUSER, J. AND WAWRZYNEK, J. 1997. Garp: A MIPS processor with a reconfigurable coprocessor. In *Proceedings of the Symposium on FPGAs for Custom Computing Machines (FCCM)*, 12–21.

HENKEL, J. AND ERNST, R. 1997. A hardware/software partitioner using a dynamically determined granularity. In *Proceedings of the Design Automation Conference (DAC)*, 691–696.

KEANE, J., BRADLEY, C., AND EBELING, C. 2004. A compiled accelerator for biological cell signaling simulations. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA)*, 233–241.

LEE, C., POTKONJAK, M., AND MANGIONE-SMITH, W. 1997. MediaBench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 330–335.

LYSECKY, R., COTTERELL, S., AND VAHID, F. 2004a. A fast on-chip profiler memory using a pipelined binary tree. *IEEE Trans. Very Large Scale Integration (TVLSI) 12*, 1, 120–122.

LYSECKY, R. AND VAHID, F. 2004. A configurable logic architecture for dynamic hardware/software partitioning. In *Proceedings of the Design Automation and Test in Europe Conference (DATE)*, 10480.

LYSECKY, R. AND VAHID, F. 2003. On-Chip logic minimization. In *Proceedings of the Design Automation Conference (DAC)*, 334–337.

LYSECKY, R., VAHID, F., AND TAN, S. 2005. A study of the scalability of on-chip routing for just-in-time FPGA compilation. In *Proceedings of the Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 57–62.

LYSECKY, R., VAHID, F., AND TAN, S. 2004b. Dynamic FPGA routing for just-in-time FPGA compilation. In *Proceedings of the Design Automation Conference (DAC)*, 954–959.

MALIK, A., MOYER, B., AND CERMAK, D. 2000. A low power unified cache architecture providing power and performance flexibility. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*, 241–243.

MARQUARDT, A., BETZ, V., AND ROSE, J. 2000. Speed and area trade-offs on cluster-based FPGA architectures. *IEEE Trans. Very Large Scale Integration Syst. (TVLSI) 8*, 1, 84–93.

MATSUMOTO, C. 2000. Triscend adds 32-bit configurable SoC line. *EE Times*, http://www.eet.com/story/OEG20000828S0015.

MEMIK, G., MANGIONE-SMITH, W., AND HU, W. 2001. NetBench: A benchmarking suite for network processors. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, 39–42.

MITTAL, G., ZARETSKY, D., TANG, X., AND BANERJEE, P. 2004. Automatic translation of software binaries onto FPGAs. In *Proceedings of the Design Automation Conference (DAC)*, 389–394.

MORRIS, K. 2005. Cray goes FPGA. *FPGA and Programmable Logic J.*, April.

PRESS, W., FLANNERY, B., TEUKOLSKY, S., AND VETTERLING, W. 1992. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, New York.

SINGH, S., ROSE, J., CHOW, P., AND LEWIS, D. 1992. The effect of logic block architecture on FPGA performance. *IEEE J. Solid-State Circuits. 27*, 3, 3–12.

STITT, G., LYSECKY, R., AND VAHID, F. 2003. Dynamic hardware/software partitioning: A first approach. In *Proceedings of the Design Automation Conference (DAC)*, 250–255.

STITT, G. AND VAHID, F. 2005. New decompilation techniques for binary-level co-processor generation. In *Proceedings of the International Conference on Computer Aided Design (ICCAD)*.

STITT, G. AND VAHID, F. 2002. Hardware/software partitioning of software binaries. In *Proceedings of the International Conference on Computer Aided Design (ICCAD)*, 164–170.

STITT, G., VAHID, F., MCGREGOR, G., AND EINLOTH, B. 2005. Hardware/Software partitioning of software binaries: A case study of H.264 decode. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 285–290.

TENSILICA, INC. 2006. XPRES compiler, automatically generate processors from standard C code. http://www.tensilica.com/products/xpres.htm.

TRISCEND CORP. 2003. http://www.triscend.com.

VENKATARAMANI, G., NAJJAR, W., KURDAHI, F., BAGHERZADEH, N., AND BOHM, W. 2001. A compiler framework for mapping applications to a coarse-grained reconfigurable computer architecture. In *Proceedings of the Conference on Compiler, Architecture, and Synthesis for Embedded Systems (CASES)*, 116–125.

VISSERS, K. 2004. Programming models and architectures for FPGAs. Keynote talk. In *Proceedings of the Conference on Compiler, Architecture and Synthesis for Embedded Systems (CASES)*.

XILINX, INC. 2006. http://www.xilinx.com.

XILINX, INC. 2005a. Customer success stories, http://www.xilinx.com/company/success/csprod.htm#embedded.

XILINX, INC. 2005b. Virtex-4 FPGAs, http://www.xilinx.com/products/silicon_solutions/fpgas/virtex/virtex4/index.htm.

XILINX, INC. 2004a. Partnering for success, Xilinx and photonic bridges. http://www.xilinx.com/ipcenter/processor_central/embedded/success_PB.pdf.

XILINX, INC. 2004b. Virtex-II Pro/ProX FPGAs, http://www.xilinx.com/products/silicon_solutions/fpgas/virtex/virtex_ii_pro_fpgas/.

XILINX, INC. 2000a. Xilinx introduces high level language compiler for Virtex FPGAs. Xilinx Press Release. http://www.xilinx.com/prs_rls/00119_forge.htm.

XILINX, INC. 2000b. Xilinx Version 3.3i software doubles clock frequencies. Xilinx Press Release. http://www.xilinx.com/prs_rls/00118_3_3i.htm.

ZAGHA, M., B. LARSON, S. TURNER, AND M. ITZKOWITZ. 1996. Performance analysis using the MIPS R10000 performance counters. In *Proceeding of the Conference on Supercomputing*, article no. 16.

ZHANG, X., WANG, Z., GLOY, N., CHEN, J. B., AND SMITH, M. D. 1997. System support for automatic profiling and optimization. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*, 15–26.

ZILLES, C. B. AND SOHI, G. S. 2001. A programmable co-processor for profiling. In *Proceedings of the International Symposium on High-Performance Computer Architectures*, 241.

Received November 2005; revised March 2006; accepted March 2006