# A Configurable Logic Architecture for Dynamic Hardware/Software Partitioning

Roman Lysecky, Frank Vahid*
Department of Computer Science and Engineering
University of California, Riverside
{rlysecky, vahid}@cs.ucr.edu
*Also with the Center for Embedded Computer Systems at UC Irvine

## Abstract

*In previous work, we showed the benefits and feasibility of having a processor dynamically partition its executing software such that critical software kernels are transparently partitioned to execute as a hardware coprocessor on configurable logic – an approach we call warp processing. The configurable logic place and route step is the most computationally intensive part of such hardware/software partitioning, normally running for many minutes or hours on powerful desktop processors. In contrast, dynamic partitioning requires place and route to execute in just seconds and on a lean embedded processor. We have therefore designed a configurable logic architecture specifically for dynamic hardware/software partitioning. Through experiments with popular benchmarks, we show that by specifically focusing on the goal of software kernel speedup when designing the FPGA architecture, rather than on the more general goal of ASIC prototyping, we can perform place and route for our architecture 50 times faster, using 10,000 times less data memory, and 1,000 times less code memory, than popular commercial tools mapping to commercial configurable logic. Yet, we show that we obtain speedups (2x on average, and as much as 4x) and energy savings (33% on average, and up to 74%) when partitioning even just one loop, which are comparable to commercial tools and fabrics. Thus, our configurable logic architecture represents a good candidate for platforms that will support dynamic hardware/software partitioning, and enables ultra-fast desktop tools for hardware/software partitioning, and even for fast configurable logic design in general.*

## Keywords

Hardware/software partitioning, FPGA fabric, configurable logic, synthesis, place and route, platforms, system-on-a-chip, dynamic optimization, codesign, self-improving chips, just-in-time compilation, warp processors, reconfigurable computing.

## 1. Introduction

Dynamic software optimization is becoming an increasingly popular method of improving software performance and power. In dynamic optimization, a user executes a standard binary program on a processor system. The processor system itself then monitors the executing binary, detects the frequently executed kernels, and optimizes those kernels. Existing optimizations include dynamic recompilation and caching of previous binary translation results [3][12][18]. Dynamic optimizations can be carried out by extra tasks sharing the processor and/or by extra hardware.

One advantage of dynamic optimization is that dynamic optimization fits seamlessly into traditional software design flows, requiring no special desktop tools and no special profiling step. While designers familiar with hardware design flows may not mind an extra tool or profiling step, the vast majority of software design flows are solidly established and not open to such changes. Further advantages include that dynamic optimization can be based on real runtime data, large amounts of such data, and changing runtime data, all of which represent additional benefits compared to desktop-based optimization. A drawback is that the optimization algorithms may have to be less powerful than desktop algorithms. Nevertheless, dynamic optimizations continue to increase in popularity.

Current dynamic software optimizations typically exhibit performance and power improvements on the order of 10-20%. However, with the advent of single-chip platforms having both microprocessors and configurable logic on the same chip, like Triscend's E5 and A7 [29] platform, Altera's Excalibur [1], Atmel's Field Programmable System Level Integrated Circuit (FPSLIC) [2], and Xilinx's Virtex-II Pro [33], a far more powerful optimization has become possible. Re-implementing the software kernels as a hardware coprocessor on the configurable logic, known as hardware/software partitioning, can result in overall software speedups of 200%-1000% [4][9][10][11][17][30], as well as reducing system energy [15][16][26][31].

Until recently, hardware/software partitioning has only been implemented as a desktop CAD tool, typically incorporated into a software compiler that partitions high-level code, like C or C++. Recently, we showed [27] that desktop hardware/software partitioning could be done starting from binaries rather than from high-level code, with competitive resulting performance and energy. Additionally, a recently introduced commercial tool performs coprocessor synthesis from standard software binaries [8]. Binary-level partitioning approaches can produce excellent results by using decompilation techniques to retrieve most of the high-level information typically lost at the binary level [7]. Such binary-level partitioning opens the door to dynamic hardware/software partitioning, in which an executing binary is dynamically optimized by moving software kernels to configurable logic – a process we call *warp processing*, since performance and energy are automatically warped during software execution.

Given the critical kernels of an application, dynamic partitioning requires decompilation, compiler optimization, behavioral synthesis, logic synthesis, and finally placement and routing onto a configurable logic architecture. While implementing all those tools on-chip for dynamic execution on a lean processor may at first sound absurd given the long computation times and huge resource usage of those tools' desktop counterparts, we have previously shown the feasibility of implementing many such tools on-chip [20][21][25]. The key is to recognize that dynamic tools need only focus on speeding up kernels, which typically consist of only a few dozen lines of code and result in hardware consisting of only 10,000 to 30,000 gates. Furthermore, dynamic tools map to only one target technology. In contrast, desktop tools must handle much bigger designs and must be much more general.

In [25], we presented the benefits and feasibility of dynamic partitioning using prototype tools, executing on a lean embedded processor and producing good results for a number of popular embedded system benchmarks. However, our earlier work in dynamic hardware/software partitioning used a very basic configurable logic architecture as a proof-of-concept. That architecture only supported combinational logic, could only implement loops with sequential memory accesses, and incurred a large routing overhead. Nevertheless, place and route was still the most computationally expensive step (as is also true using desktop tools) in dynamic partitioning – an order of magnitude more expensive than all the other steps combined. Therefore, we set out to design a new configurable logic architecture and underlying configurable logic fabric, along with a modified place and route algorithm, that together would support a much larger range of benchmarks while requiring reasonable computation time and memory resources.

In this paper, we present our warp configurable logic architecture (WCLA) for dynamic hardware/software partitioning, specifically targeted at speeding up critical loops of embedded systems applications. We did so in part by evaluating digital signal processors (DSP) and incorporating into our architecture features from the DSP domain specifically designed at increasing loop performance, such as data address generators and loop control hardware. Additionally, we analyzed potential architectural features of our configurable logic architecture with regards to the impacts on place and route tools.

In this paper, we summarize related work, introduce our WCLA, and provide results comparing our tools and architecture to a commercial tool and configurable logic, showing that we obtain similar speedups and energy savings, yet use orders of magnitude less runtime, data memory, and code memory.

## 2. Previous Work

Many configurable logic architectures have been developed to increase embedded software performance. These approaches use Field-Programmable Gate Arrays (FPGAs), reconfigurable computing, or even custom ASIC processors to achieve improvements in software execution. Existing approaches for improving embedded software performance can be classified as general (FPGAs), fine-grained configurable systems, coarse-grained configurable system, and custom ASIC processors.

Many techniques have been proposed for hardware/software partitioning. One common method for implementing hardware in such an approach is using traditional commercially available FPGAs. However, traditional FPGAs are not well suited for use in dynamic hardware/software partitioning. Traditional FPGAs are typically designed to handle an extremely wide variety of designs and are frequently used to prototype ASIC circuits. To support these vastly different designs, FPGA vendors, such as Xilinx [33] and Altera [1], design FPGAs with complex logic cells having embedded sequential components, large routing resources, large input/output resources, capabilities to support sequential logic, etc. However, in a dynamic hardware/software partitioning approach, traditional FPGAs provide more capabilities than are needed. A configurable logic architecture for implementing critical loops typically has a very simple interface to the main processor and memory and thus does not require general input/output capabilities of traditional FPGAs. Critical loops often consist of simple combinational logic or small sequential circuits and thus do not require large numbers of logic cells. Furthermore, due to their complexity, traditional FPGAs require complex synthesis, technology mapping, and place and route tools, which are not targeted for very fast execution.

Many researchers have developed techniques using FPGAs or fine-grained configurable logic in ways to improve software performance through partitioning. DISC is a run-time configurable system that dynamically swaps in hardware regions into an FPGA when needed during software execution [32]. Chimaera is a similar approach that uses an FPGA as a coprocessor tightly integrated into a processor's datapath [13]. The Garp project couples an extended MIPS processor with a reconfigurable coprocessor under direct control of the software executing on the processor [14]. Although fine-grained configurable systems have shown very good speedups, their use in a dynamic hardware/software partitioning is limited by their reliance upon complex FPGA architectures. Furthermore, with respect to improving performance of critical loops, approaches that tightly integrate the configurable logic within the datapath are not able to eliminate loop overhead, consisting of branches, comparisons, and memory address calculations.

Other approaches for increasing embedded software performance rely upon coarse-grained configurable logic architecture. MorphoSys is a reconfigurable computing platform that incorporates a RISC processor with an array of reconfigurable processing components [19]. The configurable components are coarse-grained ALU-like components that perform operations including two-operand logic functions, arithmetic functions, and multiply-accumulate.

For accelerating the execution of critical software loops, coarse-grained configurable logic architectures are limited in the number of applications that can be mapped to them. To overcome the limitations of coarse-grained configurable logic, some approaches have proposed using heterogeneous configurable logic consisting of coarse-grained units along with fine-grained configurable logic. The Chameleon [24] project and the Pleiades [31] project both propose using heterogeneous configurable logic in conjunction with a general-purpose processor. These approaches benefit from using custom designed coarse-grained units to handle commonly used operations while supporting custom operations, such as bit manipulation, using an FPGA. However, we cannot include coarse-grained functional units to support all operations frequently found within critical loops. Hence, in developing a configurable logic architecture for warp processing, specifically targeting speeding up critical loops, we must carefully analyze the possible inclusion of any coarse-grained units.

Tensilica has developed the Xtensa architecture that allows designers to customize the Xtensa processor by adding custom instructions using the Tensilica Instruction Extension (TIE) language [28]. After describing the newly added instructions, a designer is provided with a synthesizable description of the extended processor along with the associated software development tools. However, for optimizing the performance of critical software loops within embedded applications, the Xtensa processor suffers from the same drawbacks of tightly integrated fine-grained configurable logic approaches.

**Table 1**: Routability *(Routable)* and percent routing resources used (*% Routing)* for the original CLA and our WCLA.

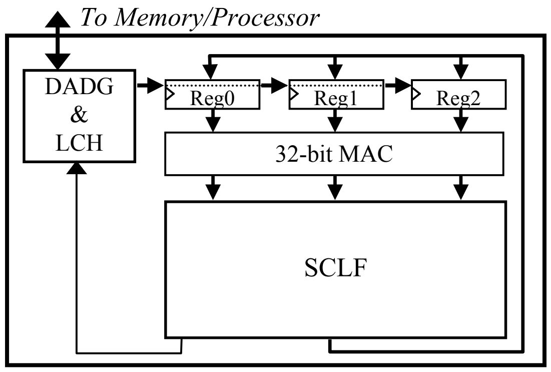
| **Benchmark** | **Original CLA** | | **SCLF** | |
|---|---|---|---|---|
| | **Routable** | **% Routing** | **Routable** | **% Routing** |
| brev | yes | 67% | yes | 12% |
| g3fax1 | yes | 76% | yes | 18% |
| g3fax2 | yes | 40% | yes | 14% |
| url | yes | 40% | yes | 14% |
| logmin | yes | 63% | yes | 14% |
| pktflow | no | | yes | 14% |
| canrdr | no | | yes | 14% |
| bitmnp | no | | yes | 12% |
| tblook | no | | yes | 47% |
| ttsprk | no | | yes | 47% |
| matrix01 | no | | yes | 61% |
| idctrn01 | no | | yes | 22% |
| g721 | no | | yes | 9% |
| **Average:** | | **57%** | | **23%** |

## 3. Configurable Logic Architecture for Dynamic Hardware/Software Partitioning

Our original dynamic-partitioning-oriented configurable logic architecture (CLA) presented in [25] incorporated a configurable logic fabric comprised of 3-input 2-output lookup tables surrounded by routing resources. While the original CLA was capable of supporting several embedded applications, the amount of routing resources used for those applications was quite large. Furthermore, some benchmarks with larger critical loops would not route using that architecture. Table 1 presents the routability and percent of routing resources used for 13 embedded benchmarks from NetBench, MediaBench, EEMBC, and Powerstone for our original CLA. While the original CLA was developed as a proof-of-concept, the routability of such an architecture is limited. The original CLA only supports five of the 13 embedded benchmarks, and on average routing for the five routable benchmarks requires 57% of the total routing resources available. Hence, we see a need to develop a configurable logic architecture and underlying configurable logic fabric that can support a larger range of applications while being simple enough to allow for lean on-chip synthesis and place and route tools.

Figure 1 shows the overall organization of our proposed warp-processor configurable logic architecture (**WCLA**) for dynamic hardware/software partitioning, 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 simple configurable logic fabric (SCLF). Our configurable logic architecture handles all memory accesses to and from the configurable logic using the data address generator, which is capable of generating addresses for up to three distinct arrays. 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 the inputs to our configurable logic fabric and can be mapped as inputs to the 32-bit (MAC) or directly mapped to the configurable logic fabric. Finally, we connect the outputs from our configurable logic fabric as inputs to the three registers using a dedicated bus.

**Figure 1:** Warp-processor configurable logic architecture (WCLA) for dynamic hardware/software partitioning.



Since we are targeting critical loops that usually iterate many times before completion, our WCLA must be able to access memory and to control the execution of the loop. One approach to handle memory accesses and loop control is to implement a finite state machine (FSM). However, using a FSM would require a configurable logic fabric supporting sequential logic circuits and would further require more complex synthesis, technology mapping, and place and route tools that must now consider scheduling and timing constraints. Instead, we found that DSP processors typically contain data address generators and loop control hardware to achieve a zero loop overhead, meaning that cycles are not wasted computing loop bounds and sequential memory addresses. We include a DADG with LCH in our warp configurable logic architecture to handle all memory accesses as well as to control the execution of the loop. However, our DADG is restricted to memory accesses that follow a regular access pattern.

Loop control hardware found within DSPs typically is 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 the loop that terminates the loop's execution. For example, in a C/C++ implementation to perform lookup with 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 our WCLA will control the loop's iterations assuming a predetermined number of iterations, but allows for terminating the loop's execution using an output from the configurable logic fabric.

As mentioned earlier, we must carefully analyze the addition of any coarse-grained hardware components within our warp configurable logic architecture. We found that there are many operations typically seen within critical software loops, including addition, subtraction, multiplication, etc. Most of these operations are easily implemented using fine-grained configurable logic. However, multipliers that operate within a single cycle are large and require many interconnects within the internal components. Furthermore, while we often see multiplications in 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 32-bit multiplier-accumulator within our configurable logic architecture to help conserve resources while reducing technology mapping and place and route execution times required for multipliers.

Figure 2(a) shows our SCLF consisting of an array of combinational logic blocks (CLB) surrounded by switch matrices (SM) 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)* or to a SM two rows apart vertically or two columns apart horizontally *(represented as dashed lines)*.

Choosing the proper size for the CLBs is important, as the CLB size directly impacts area resources and delays within our configurable logic fabric. Several studies have analyzed the impacts of CLB size on both area and timing [6][23]. These studies have shown that look-up tables (LUT) with five or six inputs result in circuits with the best performance, and LUTs with less than three inputs result in significantly worse performance. Another study analyzed the impacts on cluster sizes of CLBs on speed and area of various circuits [22]. The cluster size of a CLB is the number of single output LUTs with the CLB. Their findings indicate that cluster sizes of 3 to 20 LUTs were feasible, and a cluster size of eight produced the best tradeoff between area and 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 during dynamic partitioning, which in turn requires more complex technology mapping and placement algorithms to handle the added complexity.

Figure 2(b) shows our combinational logic block architecture. Each CLB consists of two 3-input 2-output LUTs, which provides the equivalent of a CLB consisting of four 3-input single output LUTs, and therefore should exhibit a reasonable trade-off between area and delay. We chose 3-input 2-output LUTs to simplify our technology mapping and placement algorithms by restricting the choices our tools will analyze in determining 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.

Finally, Figure 2(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 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. 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 much further apart and often include routing channels spanning an entire row or column. While such routing resources are beneficial in terms of creating compact designs with less routing overhead, the flexible routing resources require complex place and route tools that are not amenable to on-chip execution. Therefore, we chose to limit the complexity of routing resources to allow for simplified place and route algorithms.

Finally, we developed a set of dynamic hardware/software partitioning tools, the Riverside On-Chip Partitioning (**ROCPAR**) tools, including synthesis, technology mapping, placement, and routing, based on the algorithms presented in [25] for our WCLA. While our tools still incorporate the same greedy algorithms of the original tools, we updated the algorithms to take advantage of the improved routing resources and larger CLBs within our SCLF. By designing a configurable logic architecture specifically for dynamic hardware/software partitioning, we have expanded the range the applications that a dynamic hardware/software partitioning approach can support. As shown in Table 1, our WCLA uses much less routing resources, using an average of 22% of the routing resources available for the given examples. Furthermore, for the five examples supported by the original architecture, our WCLA uses on average only 14% of the available resources, which corresponds to a 4X improvement over the original CLA, for the same silicon area.

## 4. Results

We compare a dynamic hardware/software partitioning using our WCLA with a typical hardware/software partitioning approach targeting a Xilinx Virtex-E FPGA, comparing speedup and energy reduction for 13 embedded systems benchmarks from NetBench, MediaBench, EEMBC, and Powerstone. Our experimental framework consists of an ARM7 processor executing at 75 MHz coupled with either our WCLA or a Xilinx Virtex-E series FPGA. Furthermore, our WCLA executes at a fixed frequency of 60 MHz due to the current implementation of our MAC and DADG, while

**Figure 2:** (a) Warp processing simple configurable logic fabric, (b) Combinational logic block, and (c) Switch matrix architecture.

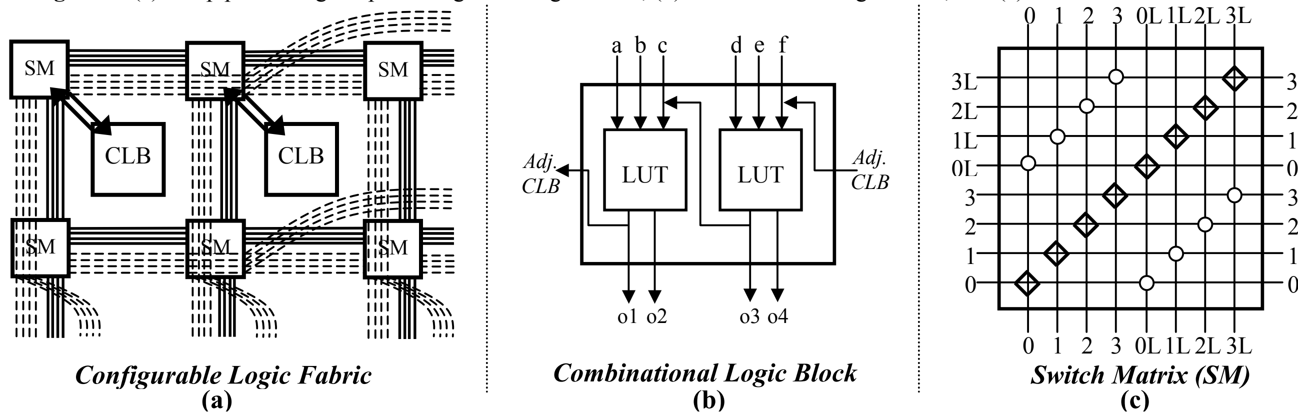


**Figure 3:** Speedups for HW/SW partitioning using our WCLA and a Xilinx Virtex-E FPGA.

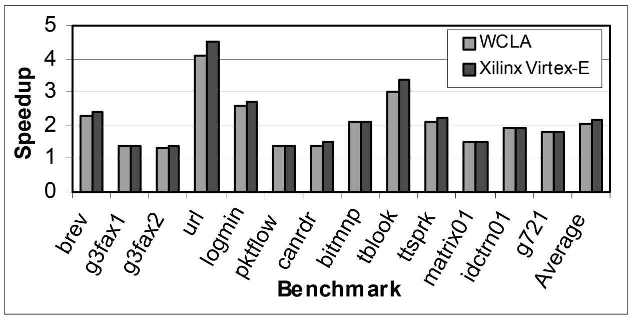
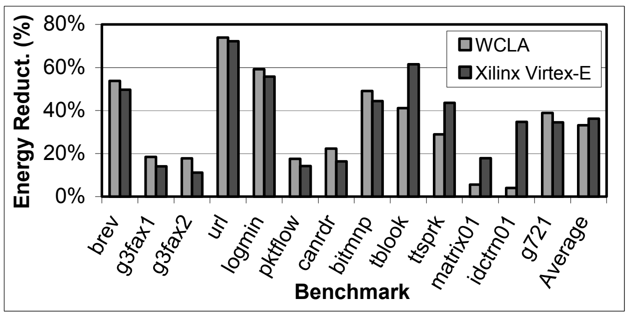


**Figure 4:** Percent energy reduction for HW/SW partitioning using our WCLA and a Xilinx Virtex-E FPGA.



the FPGA executes at the highest frequency possible for each design when synthesized and mapped using Xilinx ISE 4.1 [33]. For each benchmark, we determined the single most critical loop and partitioned the critical loop to hardware either using our ROCPAR or synthesizing a custom VHDL implementation of the critical loop. While partitioning a single critical loop produces good results, the speedups would be even greater if we considered multiple loops.

Figure 3 and Figure 4 highlight the speedup and energy reduction of our WCLA and the Xilinx FPGA for all 13 benchmarks. In determining speedups of the two approaches, all software execution times were determined using the SimpleScalar simulator [5] ported for the ARM instruction set. Additionally, we determined execution times for the hardware implementations using a high-level simulator for our configurable logic architecture and using VHDL simulations with the appropriate clock frequency for each implementation determined during synthesis. We calculated the energy required for each partitioned application using the equations in Figure 5. We used the Xilinx Virtex Power Estimator along with information provided by Xilinx ISE to determine total power consumed by the FPGA when active as well as the overall static power. The approach used by the Xilinx Power Estimator consists of providing information including the number of LUTs, number of flip-flops, average switching within the FPGA, and clock frequency to determine the power consumed by the FPGA. We implemented a similar estimation approach to determine the power consumed by our WCLA. We implemented a small version of our configurable logic architecture in VHDL and synthesized the design using Synopsys Design Compiler targeting the UMC 0.18 µm technology library provided by Artisan Components. Using gate-level simulations, we determined the power consumed by individual components within our configurable logic architecture.

Although our WCLA is much simpler than the Xilinx FPGA, on average our WCLA achieved a speedup of 2.1 with an average energy reduction of 33.1%. These results are very close to the average speedup of 2.2 and energy reduction of 36.2% achieved using a Xilinx FPGA. We initially thought that while our configurable logic architecture would not produce results better than a general FPGA, our configurable logic architecture should result in less overall energy consumption compared with an FPGA. However, for several benchmarks, including *tblook*, *ttsprk*, *matrix01*, and *idctrn01*, our WCLA had a higher energy consumption than the Xilinx FPGA. We determined that for *matrix01* and *idctrn01*, the high energy consumption was mainly caused by the use of our embedded multiplier, which has a higher energy consumption than the dedicated multiplier support within the Xilinx FPGA. Additionally, all four benchmarks had large energy consumption resulting from a large usage of routing resources, indicating the need to develop new place and route algorithms that can run on-chip environment while producing good results – a task we are working on.

We also evaluated our ROCPAR tools, comparing them with Xilinx ISE 4.1. Table 2 displays the average data memory usage and code size in kilobytes and the average execution times in seconds of Xilinx ISE and ROCPAR executing on a 1.4 GHz Pentium workstation. Table 2 also displays execution time in seconds for ROCPAR executing on a 75 MHz ARM7 processor. On average, executing our simplified tools in an embedded environment requires less than 2 seconds, which is quite feasible. Furthermore, the maximum data memory required was on average less than 10 kilobytes. While Xilinx ISE was never designed to execute on-chip, the large data memory requirements indicate that the algorithms and data structures used by these tools are not suitable for on-chip execution either. Thus, new synthesis, technology mapping, and place and route tools are required for a dynamic hardware/software partitioning approach.

## 5. Conclusions

Dynamic hardware/software partitioning represents a far more powerful dynamic optimization than currently proposed dynamic software optimizations, the former achieving 200%-400% performance improvements rather than the typical 10%-20% of the latter – with greater improvements easily possible by partitioning more than one loop. The hardest step of dynamic partitioning is placing and routing onto a configurable logic

**Figure 5:** Equation for determining energy consumption after hardware/software partitioning.

$$E_{total} = E_{ARM} + E_{HW}$$

$$E_{ARM} = P_{ARM\ (idle)} \times t_{idle} + P_{ARM\ (active)} \times t_{active}$$

$$E_{HW} = P_{HW} \times t_{active} + P_{static} \times t_{total}$$

**Table 2**: Average data memory usage *(kilobytes)*, code size *(kilobytes),* and execution time *(seconds)* of Xilinx ISE 4.1 and ROCPAR executing on a PC and a 75 MHz ARM7.

| | Data Memory | Instruction Memory | Execution Time |
|---|---|---|---|
| **Xilinx ISE (PC)** | 54384 | 58700 | 9.1 |
| **ROCPAR (PC)** | 6 | 57 | 0.2 |
| **ROCPAR (ARM7)** | 6 | 57 | 1.4 |

fabric. We have simultaneously designed a new configurable logic architecture and simple configurable logic fabric along with accompanying place and route algorithms specifically intended for dynamic partitioning. Our architecture includes data address generators for accessing up to three distinct arrays in memory, loop control hardware to control loop iterations, a 32-bit multiplier-accumulator, and a configurable logic fabric with simple combinational logic blocks and routing resources. We have shown that our architecture, fabric, and accompanying algorithms result in place and route that is 50 times faster than commercial tools, using 10,000 times less data memory and 1,000 times less code memory, and running in a reasonable time of just a few seconds on a small embedded microprocessor. Yet, we also showed that our architecture and lean tools still obtain comparable speedups to commercial configurable logic and tools, with an average software speedup of 2.1 and energy savings of 33% when partitioning even just one loop – speedups and savings will be even more when additional loops are considered. We are continuing to improve our architecture, fabric, and tool set, to handle a larger set of applications.

## 6. Acknowledgements

This research was supported in part by the National Science Foundation (CCR-0203829, CCR-9876006) and by the Semiconductor Research Corporation (grant CSR 2002-RJ-1046G).

## 7. References

[1] Altera Corp. http://www.altera.com, 2003.

[2] Atmel Corp. http://www.atmel.com, 2003.

[3] Bala, V., E. Duesterwald, S. Banerjia. Dynamo: A Transparent Dynamic Optimization System. Conf. on Programming Language Design and Implementation, 2000.

[4] Balboni, A., W. Fornaciari and D. Sciuto. Partitioning and Exploration in the TOSCA Co-Design Flow. International Workshop on Hardware/Software Codesign, pp. 62-69, 1996.

[5] Burger, D., T. Austin. The SimpleScalar Tool Set, version 2.0. SIGARCH Computer Architecture News, Vol. 25, No. 3, 1997.

[6] Chow, P., S. Seo, J. Rose, K. Chung, G. Paez-Monzon, I. Rahardja. The Design of an SRAM-Based Field-Programmable Gate Array, Part I: Architecture. IEEE Transactions on VLSI Systems, 1999.

[7] Cifuentes, C., M. Van Emmerik, D.Ung, D. Simon, T. Waddington. Preliminary Experiences with the Use of the UQBT Binary Translation Framework. Proceedings of the Workshop on Binary Translation, 1999.

[8] Critical Blue, http://www.criticalblue.com, 2003.

[9] Eles, P., Z. Peng, K. Kuchchinski and A. Doboli. System Level Hardware/Software Partitioning Based on Simulated Annealing and Tabu Search. Kluwer's Design Automation for Embedded Systems, vol2, no 1, pp. 5-32, Jan 1997.

[10] Ernst, R., J. Henkel, T. Benner. Hardware-Software Cosynthesis for Microcontrollers. IEEE Design & Test of Computers, pages 64-75, October/December 1993.

[11] Gajski, D., F. Vahid, S. Narayan and J. Gong. SpecSyn: An Environment Supporting the Specify-Explore-Refine Paradigm for Hardware/Software System Design. IEEE Trans. on VLSI Systems, Vol. 6, No. 1, pp. 84-100, 1998.

[12] Gschwind, M., E. Altman, S. Sathaye, P. Ledak, D. Appenzeller. Dynamic and Transparent Binary Translation. IEEE Computer, Vol. 3, pp.70-77, 2000.

[13] Hauck, S., T. Fry, M. Hosler, J. Kao. The Chimaera Reconfigurable Functional Unit. FPGAs for Custom Computing Machines (FCCM), pp. 87-96, 1997.

[14] Hauser, J., J. Wawrzynek. Garp: A MIPS processor with a reconfigurable coprocessor. IEEE Symposium on FPGAs for Custom Computing Machines (FCCM), 1997.

[15] Henkel, J. A low power hardware/software partitioning approach for core-based embedded systems. Design Automation Conference, pp. 122 – 127,1999.

[16] Henkel, J., Y. Li. Energy-conscious HW/SW-partitioning of embedded systems: A Case Study on an MPEG-2 Encoder. Proceedings of Sixth International Workshop on Hardware/Software Codesign, March 1998, pp. 23-27.

[17] Henkel, J., R. Ernst. A Hardware/Software Partitioner using a Dynamically Determined Granularity. Design Automation Conference, 1997.

[18] Klaiber, A. The Technology Behind Crusoe Processors. Transmeta Corporation White Paper, 2000.

[19] Lee, M., H. Singh, G. Lu, N. Bagherzadeh, F. Kurdahi, E. Filho, V. Alves. Design and Implementation of the MorphoSys Reconfigurable Computing Processor. Journal of VLSI Signal Processing-Systems for Signal, Image and Video Technology, 2000.

[20] Lysecky, R., F. Vahid. A Codesigned On-chip Logic Minimizer. International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2003.

[21] Lysecky, R., F. Vahid. On-chip Logic Minimization. Design Automation Conference (DAC), 2003.

[22] Marquardt, A., V. Betz, J. Rose. Speed and Area Trade-offs on Cluster-based FPGA Architectures. IEEE Trans. on VLSI, 2000.

[23] Singh, S., J. Rose, P. Chow, D. Lewis. The Effect of Logic Block Architecture on FPGA Performance. IEEE Journal of Solid-State Circuits, Vol. 27, No. 3, 1992.

[24] Smit, G., P. Havinga, L. Smit, P. Heysters, M. Rosien. Dynamic Reconfiguration in Mobile Systems. Proc. FPL, 2002.

[25] Stitt, G., R. Lysecky, F. Vahid. Dynamic Hardware/Software Partitioning: A First Approach. Design Automation Conference (DAC), 2003.

[26] Stitt, G. and F. Vahid. The Energy Advantages of Microprocessor Platforms with On-Chip Configurable Logic. IEEE Design and Test of Computers, Nov/Dec 2002.

[27] Stitt, G., F. Vahid. Hardware/Software Partitioning of Software Binaries. IEEE/ACM International Conference on Computer Aided Design (ICCAD), 2002.

[28] Tensilica, Inc. http://www.tensilica.com, 2003.

[29] Triscend Corp. http://www.triscend.com, 2003.

[30] Venkataramani, G., W. Najjar, F. Kurdahi, N. Bagherzadeh, W. Bohm. A Compiler Framework for Mapping Applications to a Coarse-grained Reconfigurable Computer Architecture. Conf. on Compiler, Architecture and Synthesis for Embedded Systems (CASES 2001), 2001.

[31] Wan, M., Y. Ichikawa, D. Lidsky, L. Rabaey. An Energy Conscious Methodology for Early Design Space Exploration of Heterogeneous DSPs. Proc. ISSS Custom Integrated Circuits Conference (CICC).

[32] Wirthlin, M., B. Hutchings. DISC: The Dynamic Instruction Set Compiler. IEEE Symposium on FPGAs for Custom Computing Machines (FCCM), 1995.

[33] Xilinx, Inc. http://www.xilinx.com, 2003.