Creating HW/SW Co-Designed MPSoPC's from High Level Programming Models

Eugene Cartwright, Sen Ma, David Andrews, and Miaoqing Huang
Computer Science & Computer Engineering Department, University of Arkansas
{eugene,senma,dandrews,mqhuang}@uark.edu

ABSTRACT

FPGA densities have continued to follow Moore's law and can now support a complete multiprocessor system on programmable chip. The benefits of the FPGA include the ability to build a customized MPSoC system consisting of heterogeneous processing resources, interconnects and memory hierarchies that best match the requirements of each application. In this paper we outline a new approach that allows users to drive the generation of a complete hardware/software co-designed multiprocessor system on programmable chip from an unaltered standard high level programming model. We use OpenCL as our specification framework and show how key API's are extracted and used to automatically create a distributed shared memory multiprocessor system on chip architecture for Xilinx FPGA's. We show how OpenCL API's are easily translated to hthreads, a hardware-based microkernel operating system to provide pthreads compliant run time services within the MPSoPC architecture.

KEYWORDS: Heterogeneous computing, Operating systems, Uniform programming model.

1. INTRODUCTION

Field-Programmable Gate Array (FPGA) densities have continued to track Moore's law. Xilinx's latest generation of FPGAs, termed *platform* FPGAs contain sufficient gates, block RAMs (BRAMs), and diffused logic to support a complete Multiprocessor System on Programmable Chip (MPSoPC). To exploit this new capability, Xilinx is encouraging designers to target the processor instead of the transistor as the smallest design quantum [1]. Targeting processors instead of transistors as the lowest level design quantum promises many advantages for widening the use of FPGA technologies, lowering design costs and time to market, and increasing designer productivity. While this approach sacrifices peak performance for productivity, switching focus from low-level hardware circuit design to writing high-level parallel programs makes FPGAs accessible to the broad base of software developers who do not possess hardware design skills. Both software and hardware designers can benefit from adopting familiar software development tools and run time systems to reduce design times, and enable code portability and reuse. These should be welcome advantages within the FPGA design community that largely worked with tools and design flows that result in relatively inefficient designer productivity [2].

Three barriers must be overcome to make this vision a reality. First, the selection of available soft IP processor components must continue to expand, and with appropriate *compiler* support. Efforts such as eMIPS and Vespa [3, 4] are increasing the selection of customizable, and vector and array programmable processors that can be integrated into a MPSoPC platform FPGA. These efforts cannot be fully exploited until associated advancements in compiler technology for heterogeneous systems are also achieved.

The second barrier is the inability to simply adopt our existing commodity operating systems, which themselves are undergoing a major paradigm shift to meet the needs of next generation homogeneous and heterogeneous multicores. The structure of our monolithic kernels were never created with scalability in mind, and integrating processors with different Instruction Set Architectures (ISA's) breaks current operating systems' abilities to provide a single uniform set of abstract services that *efficiently* execute across mixes of processor ISAs. Resolving this issue is pivotal for platform MPSoPCs as the operating system forms the foundation upon which higher level run time services and user code bases are built.

The third barrier is the lack of a standard high level programming model. Within the commercial sector several new programming frameworks are emerging specifically for heterogeneous manycores. OpenMP [5] has enjoyed some success for Symmetric Multiprocessor (SMP) global memory systems. More recently OpenCL [6] has gained popularity and is now a standard for distributed memory heterogeneous architectures. For reconfigurable systems, we are interested in extending the capabilities of the high level programming to additionally drive the generation of a semi-custom heterogeneous MPSoPC. Third generation system level synthesis approaches made strides in generating hardware accelerators from computationally intensive portions of a high level language, such as C. However, translating a series of ALU operations into a more efficient custom circuit is orthogonal to creating an overarching multiprocessor system on chip architecture. We believe this new challenge represents a next, or forth generation of system level synthesis.

1.1. Contributions of This Work

The contributions of this work are on two broad fronts.

- First, we show a new automated architecture synthesis capability. We have created FSM SYS Builder, which assembles all soft IP components including processors, busses, bus bridges, memory hierarchies and DMA's specifically for a distributed shared memory architecture.
- Second, we show how threads from the OpenCL specification can be transparently compiled for multiple heterogeneous processors and linked into our pthreads compliant real time hardware microkernel operating system.

For this work, we have selected simple MicroBlaze processors to serve as our heterogeneous processor resources. At this early stage of heterogeneous multiprocessor system on programmable chip synthesis, no suitable higher performance vector processor and *compiler* exists. Without loss of generality, the MicroBlaze allows us to validate our ability to automatically assemble a distributed shared memory system with soft IP processors, link in operating system libraries, compile and run code for each processor type.

2. OPENCL PROGRAMMING MODEL

Historical trends in programming model design have continued to increase the level of abstraction for programmers

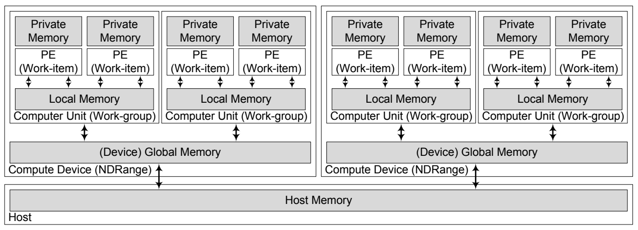

Figure 1. Memory Hierarchy in OpenCL Programming Model

to foster portability, code reuse, and productivity. OpenCL reverses this trend by reflecting architecture specific details back up through the protocol stack to the application programmer. Specifically the model recognizes how naive partitioning of data sets across distributed memory hierarchies can negate the performance benefits of parallel processors. OpenCL models a platform as consisting of a host, or master processor, and slave compute devices as shown in Figure 1. Applications are explicitly partitioned to run on each processor resource type. Code destined to run on slave processors is decomposed into multiple *kernels*, each of which is executed on a compute device during the run time. Each compute device is comprised of multiple compute units, which are composed by an array of processing elements (PEs). When a kernel is scheduled to run on a compute device, it is realized in a grid of threads. The grid and the thread are termed NDRange and work-item, respectively, in OpenCL. An NDRange is divided into work-groups, each of which is scheduled to execute on a compute unit. There is a memory hierarchy consisting of host, global, local, and private memories in OpenCL and the programmer is responsible to marshal the data between these four types of memories. More precisely, the host function needs to transfer data between the host memory and global memories explicitly. Each work-item can access its private memory, the local memory of the resident compute unit, and the global memory of the residing compute device.

3. DESIGN APPROACH

Figure 2 shows a high level description of our approach. The creation of a hardware/software co-designed system starts by extracting OpenCL API's for specifying the number and types of processors per kernel, and API's used for run time operating system services. The number and types of processors are then input into FSM SYS Builder, which integrates the processors within an architecture template derived from the OpenCL execution model. The FSM SYS Builder outputs mss and mhs files that are then exported into the Xilinx Platform Studio XPS tool. The run time

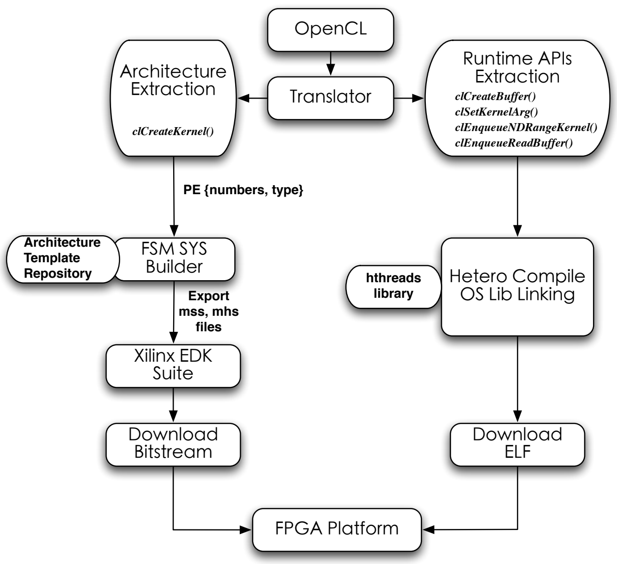

Figure 2. High Level Design Flow

API's are extracted and then linked against our hthreads operating system services libraries. To explain our approach we use the example shown in Figure 3, which shows pseudo code for two kernels, one for computing a matrix multiplication and the other for a matrix addition.

3.1. Architecture Generation

Architecture generation focuses on assembling and integrating all components necessary to form a complete heterogeneous multiprocessor system on programmable chip. We adopt the formation of architecture templates that match popular programming models. Figure 4 shows a generic architecture template produced by FSM SYS Builder consisting of three Microblaze processors, hierarchical bus structures, our hthreads hardware microkernel cores, and three tiered memory system. To create such a system, FSM SYS Builder inputs the OpenCL API `clCreateKernel()` call, which provides numbers and types of processors for each kernel. The OpenCL execution model requires a four tiered distributed memory hierarchy. In the generic architecture template a scalar CPU is implemented to realize the computer device. Therefore the *local memory* and the device global memory are combined in Figure 4. Thus we first integrate private scratchpad memory for each processor. The per processor work-items within a work-group may also access local memory/device global memory, but the host memory is only accessible from the host processor. For Xilinx based systems, we use BRAM to form the private, the local, and the device global memory memories, and DRAM for the host memory. The local memory is omitted if the compute device is a scalar processor. The amount of BRAM banks configured into private and device global memory is dependent on the total number of BRAM's contained within the device. For our existing Xilinx V5Pro devices, we allocate 16 KBytes of BRAM to each processor as private memory, and partition the remaining device dependent BRAM's evenly with each processor as device global memory. For our experimental MicroBlaze based systems, private memory are connected to each processor through Instruction Local Memory Bus (LMB) and not accessible by any other processor. We include a secondary PLB bus and connect the MicroBlaze processors and device global memories onto the bus accessible through a global address space. BRAM within Xilinx devices are dual ported with two bus interfaces. The first port of each BRAM is connected to the secondary PLB bus and can be accessed within the global address, and the second port is connected directly to each MicroBlaze to provide fast (2-cycle) access. Not shown in

Algorithm 0.1

```
The Matrix Multiplication of OpenCL:
begin
    $Matrix_A := M \times T$;
    $Matrix_B := T \times N$;
    $OpenCL\_Create\_Context$;
    $OpenCL\_Create\_Buffer(Matrix_C)$;
    $OpenCL\_Create\_Kernel(Kernel\_Matrix\_Multiplication)$;
    $OpenCL\_EnqueueNDRangeKernel(Kernel)$;
where
funct $Kernel\_Matrix\_Multiplication(A, B, C) \equiv$
    $t_x := Get\_Global\_id(x)$;
    $t_y := Get\_Global\_id(y)$;
    for $k := 1$ to $Width\_A$ step 1 do
        $item_A := A[t_y \times Width\_A + k]$;
        $item_B := B[k \times Width\_B + t_x]$;
        $sum := sum + item_A \times B$ od;
    $C[t_y \times Width\_A + t_x] := sum$.
end
```

Algorithm 0.2

```
The Matrix Addition of OpenCL:
begin
    $Matrix_A := M \times T$;
    $Matrix_B := T \times N$;
    $OpenCL\_Create\_Context$;
    $OpenCL\_Create\_Buffer(Matrix_D)$;
    $OpenCL\_Create\_Kernel(Kernel\_Matrix\_Addition)$;
    $OpenCL\_EnqueueNDRangeKernel(Kernel)$;
where
funct $Kernel\_Matrix\_Addition(A, B, D) \equiv$
    $t_x := Get\_Global\_id(x)$;
    $t_y := Get\_Global\_id(y)$;
    $item_A := A[t_y \times Width\_A + t_x]$;
    $item_B := B[t_y \times Width\_B + t_x]$;
    $sum := sum + item_A \times B$;
    $D[t_y \times Width\_A + t_x] := sum$.
end
```

Figure 3. OpenCL Specification of Sample Applications

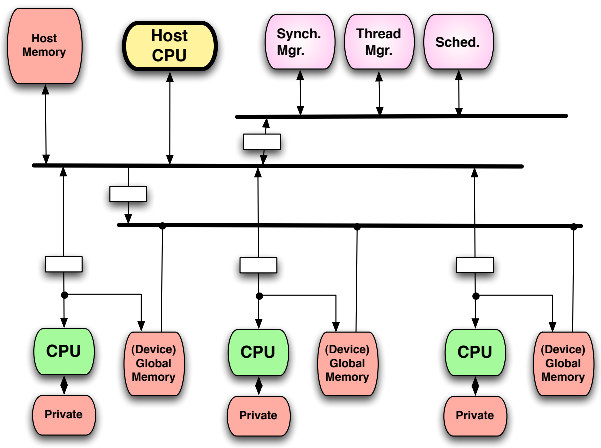

Figure 4. Architecture Template

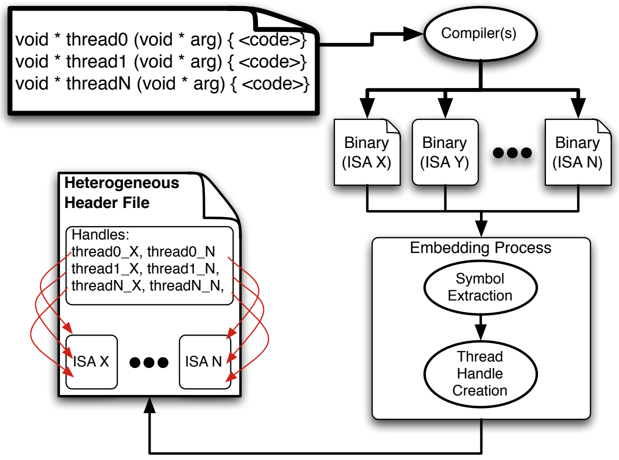

Figure 5. Embedding Process

the template but included are DMA eigens that provide fast data marshaling between host memory and device global memory. All components are assembled and linked together by FSM SYS Builder to produce the *mss* and *mhs* files to input into Xilinx's XPS. FSM SYS Builder thus frees designers from having to manually gather and integrate the system components, and provides a complete synthesizable architecture tailored for the OpenCL specification. The architecture template created by FSM SYS Builder can be used stand alone, or can be further modified within the Xilinx EDK tool suite.

3.2. Compilation flow

This work builds on our prior heterogeneous compilation and hthreads hardware microkernel infrastructure. The hthreads operating system is a pthreads compliant set of hardware/software co-designed services designed for resolving the heterogeneity and scalability issues of next generation manycore architectures. User thread application code is extracted and passed to our heterogeneous compilation infrastructure. A thin set of wrappers were written that translate each OpenCL API shown in Figure 3 into equivalent pthreads compliant hthreads API's. We created a specialized compilation technique similar to those used by IBM's Cell and Intel's EXOCHI [7–9] to embed heterogeneous binaries into a single executable image for Xilinx FPGA's. This approach has many benefits including allowing a developer to create an application as a single, coherent program rather than a set of several programs that contain implicit interactions. Our heterogeneous compilation flow allows developers to pass a thread body, library and support functions through a compiler for each target architecture. This process can be performed manually by the programmer, or can be automated using a specialized compiler and build system [9]. In our system, all heterogeneous threads are defined in separate files so that they can be easily compiled for a variety of target architectures. All executables are then embedded into a single heterogeneous executable using command line tools. Figure 5 shows the embedding process that takes place during heterogeneous compilation. The compilation flow uses an embedding process that compiles application threads for each architecture, links each binary with the operating system code, and embeds the executables into a header. The header file also contains a set of thread *handles* that are analogous to function pointers; however they allow indirection into the different executables associated with each processor type. The indirection is needed as function pointers in C are not first-class objects. During `thread_create`, the operating system determines the availability of all system processor resources and will schedule the new thread on the first available processor. This occurs transparently to the user.

3.3. Hthreads: A Hardware Microkernel

This work builds on our prior hthreads hardware microkernel infrastructure. Hthreads was designed to provide a unified set of operating system services for application programs running on scalable numbers of parallel heterogeneous processor resources. Hthreads is a pthreads compliant hardware microkernel that has migrated mutex and synchronization primitives, thread management and thread scheduling into parallel hardware cores [10].

From a performance perspective, each of the hardware components are designed to operate independently and in parallel following the microkernel philosophy. This enables the hthreads system services to achieve a significantly tighter performance envelope compared to monolithic approaches [10, 11]. Although similar to distributing operating system services across different cores, the execution time and jitter of each system service is decreased through a more efficient implementation in hardware. Multiple services can be accessed simultaneously by threads running throughout the system. This approach is better-suited for systems that seek to scale to thousands of active threads. We present a brief overview of the hthreads hardware microkernel cores. Information concerning the philosophy, implementation details, and performance comparisons for hthreads can be found in [12].

Thread Management: The thread manager provides a centralized repository of state information for each thread that is independent of the processor type or location upon which the thread is currently running. Maintaining a separate global repository of thread state eliminates the need to periodically query individual processors for updates, and serves as a database for other OS services. Individual OS services can query the hardware repository without having to interrupt and interfere with application threads executing on other processors. All thread data is stored in a centralized memory which is operated on by the thread manager. Additionally, many thread management operations are based on error-checking, which mostly involves boolean arithmetic. In hardware, this can be calculated in a single block of fast combinational logic instead of a set of sequential CPU instructions.

Thread Scheduling: Hardware based schedulers have been well understood within embedded real time systems providing low latency and low jitter tight scheduling envelopes [13]. In addition to reducing the latency of scheduling decisions, a hardware scheduler also executes independently and in parallel with applications running on system processors [14]. This allows the scheduler to predetermine when a context switch for a scheduling decision will be made *before* it occurs. The leads to low-overhead scheduling decisions that can be many orders of magnitude faster than a software-based scheduler [15]. Lower overhead and jitter during scheduling promises faster context switches, and reduces the overall system overhead incurred during OS invocations. This makes the OS more efficient, thus enabling greater number of threads, and OS invocations. A scheduler peripheral accessible by all types of processors is capable of maintaining a centralized repository containing information about each type of processor. This allows more intelligent scheduling decisions to be made, as the scheduler has global information about the system.

Thread Synchronization: Synchronization primitives, such as locks and mutexes, are very simple OS objects. Implementing mutex primitives as a hardware peripheral provides the ability for any type of processor to synchronize in a heterogeneous system; an ability not found in systems that depend on ISA-specific atomic operations [16]. Hardware-based synchronization primitives scale with the heterogeneity-level of a system, as the invocation mechanism is common to all processors. The speed at which a hardware-based synchronization mechanism operates can lead to lower overhead system calls, thus dissipating contention, and enhancing scalability. This approach also removes dependence on heavyweight cache coherence protocols for lock distribution, as the data associated with each lock is held within a centralized piece of hardware.

The operating system consists of minimal software libraries that run on each processor. The software portion of the operating system, called the hardware abstraction layer (HAL), must run on each processor to support the execution of heterogeneous threads and provide access to OS services. The HAL acts as a shallow protocol stack; translating each processor's API invocations into memory-mapped I/O requests that interact with the underlying hthreads OS cores.

4. EXPERIMENTAL RESULTS

In order to verify the correctness of the design flow in Figure 2, we use the two kernels in Figure 3. We created an OpenCL program for test purposes with the two kernels mapped to separate Microblaze scalar processors. The *translator* converts each kernel into separate threads. In all the experiments, we ran the same application 20 times and took the average. The FPGA test board is Xilinx ML507 consisting of XC5VFX70T device. The version of ISE is 10.1.

Two different platform implementations were created to evaluate the effects of the memory hierarchy. In the first platform implementation, the device global memory address space is mapped to a region in host memory. This mapping is hidden from the programmer with data access for the Microblaze to the DRAM through the PLBv4.6 bus. There are two disadvantages for this platform implementation. In the following text we refer to this platform implementation as "w/o DMA implementation". In the second platform implementation, a device global memory is generated for each Microblaze processor. Before and after the execution of a kernel, the source data and the result data are transferred from the host memory to and from device global memory through DMA engines. Although data transfer overhead is introduced, data access contention is significantly reduced resulting in performance improve-

Table 1. Performance Comparison of Matrix Addition and Multiplication

Operation	w/o DMA (μs)	w/ DMA (μs)			
		D2B*	Computation	B2D*	Total
Multiplication	17,065	*41*	*7,814*	*47*	7,903
Addition	3,904	*43*	*2,260*	*50*	2,354

*D2B: DRAM→BRAM; B2D: BRAM→DRAM

Table 2. Performance Comparison of Two Memory Hierarchies for 20 Iterations of Matrix Multiplication

Design	Computation Time (μs)		Resource Utilization (LUTs)	
	w/o DMA	w/ DMA	w/o DMA	w/ DMA
2-core	361,581	156,595	19,096 (42%)	19,920 (44%)
3-core	390,094	156,707	22,487 (50%)	23,462 (52%)
4-core	421,869	156,863	25,984 (58%)	27,241 (60%)
5-core	463,658	156,988	29,774 (66%)	30,660 (68%)
6-core	524,301	NA*	33,171 (74%)	34,007 (75%)

*Unknown Failure at Time of Publication

ments. This platform implementation is called "w/ DMA implementation".

In the first experiment we want to demonstrate the importance of the memory hierarchy. A 2-kernel OpenCL application is mapped to both with/without DMA platforms. One kernel is matrix multiplication and the other is matrix addition. The same two matrices are used for addition and multiplication, i.e., $A + B = C, A \times B = D$, and their sizes are 20×20. All the items in matrix are integer, i.e., 32-bit. The execution of these two kernels is allocated to two separate Microblaze processors so that the computation is carried out in parallel. The performance for these two kernels under both cases is listed in Table 1. Apparently, the use of the DMA in a memory hierarchy can significantly improve the performance compared with a central memory architecture. The performance of one iteration of matrix multiplication and addition is improved by $2.16\times$ and $1.66\times$ respectively. In all experiments in this work, the data cache is not selected for the Microblaze processors. Therefore each data access has to go through the PLBv4.6 bus in the "w/o DMA implementation" case. One the other hand, the latency is 2 clock cycles for accessing OBM in the "w/ DMA implementation", therefore reducing the data communication overhead substantially.

In order to test the scalability of these two different implementation options, we increase the number of kernels from 2 to 6. In the current implementation, the Microblaze fetches the instruction through the Multiple-Port-Memory-Controller (MPMC) from the off-chip DRAM. The MPMC has 8 ports in which two ports are occupied by the PowerPC processor and the PLBv4.6 bus respectively. Therefore, only 6 ports are available for Microblaze devices. In this second test, all kernels perform the matrix multiplication for 20 iterations. The results for both cases are listed in Table 2. Apparently, the hierarchical memory demonstrates an excellent scalability. When the number of Microblaze processors moves from 2 to 6, there is no performance degradation. On the other hand, the computation time climbs slightly as the number of processors increases. Although the absolute increase of computation time is not very significant due to the small number of Microblaze processors we can put into the platform, the trend is very evident. As we put more processing cores into the system in the future design, we expect that the performance of the "w/o DMA implementation" will not scale. In the meantime, the hardware resource cost for adding the DMA is trivial, as shown in Table 2.

5. CONCLUSION

FPGA densities have continued to follow Moore's law and can now support a complete multiprocessor system on programmable chip. Performance can be provided through scalable parallelism instead of custom point designed accelerators. This has the potential to open the benefits of FPGA's to the widening domain of application software developers instead of hardware designers. To tap this potential requires three new capabilities. First, the selection of programmable soft processors including compilers are required. Second, familiar protocol stacks including operating systems must be provided. Third, appropriate high level languages for heterogeneous manycores are required for use by both the programmer and to guide the automatic synthesis of a heterogeneous multiprocessor system on programmable chip. In this paper we have outlined a new approach that allows users to drive the generation of a complete multiprocessor system on programmable chip from a standard high level programming model. We targeted OpenCL as our high level language and showed how key API's can be extracted and used by FSM SYS Builder to automatically create a multiprocessor system on chip architecture for Xilinx FPGA's. We also showed how OpenCL API's are translated to hthreads, a hardware-based microkernel operating system to provide pthreads compliant run time services. Our current prototype uses simple Microblaze processors as our heterogeneous processor resources. While the Microblaze nullifies any real performance comparisons of our automatically generated with other hand crafted systems, it does allow us to verify our end to end flow. We anticipate substituting more appropriate vector type processors that will allow future performance comparisons. The results using synthetic kernels demonstrate the correctness of the proposed FSM SYS Builder and the performance scalability of the generated multiprocessor architecture.

REFERENCES

[1] S. Trimberger, "FPL 2007 Xilinx Keynote Talk - Redefining the FPGA," http://ce.et.tudelft.nl/FPL/trimbergerFPL2007.pdf, last accessed May 10, 2011.

[2] B. Nelson, "Fpga design productivity: Exiting limitations, root causes, and solutions," http://www.et.byu.edu/~nelson/FPT2008PreWorkshop/nelson_productivity.pdf, last accessed May 10, 2011.

[3] R. N. Pittman, N. L. Lynch, and A. Forin, "eMIPS, A Dynamically Extensible Processor," Microsoft Research, Tech. Rep. MSR-TR-2006-143, Oct. 2006.

[4] P. Yiannacouras, J. G. Steffan, and J. Rose, "VESPA: Portable, Scalable, and Flexible FPGA-Based Vector Processors," in *CASES '08: Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. New York, NY, USA: ACM, 2008, pp. 61–70.

[5] L. Dagum and R. Menon, "OpenMP: An Industry-Standard API for Shared-Memory Programming," *Computing in Science and Engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[6] "OpenCL - The open standard for parallel programming of heterogeneous systems," http://www.khronos.org/opencl/, last accessed May 10, 2011.

[7] P. H. Wang, J. D. Collins, G. N. Chinya, H. Jiang, X. Tian, M. Girkar, N. Y. Yang, G.-Y. Lueh, and H. Wang, "EXOCHI: Architecture and Programming Environment for a Heterogeneous Multi-Core Multithreaded System," *ACM SIGPLAN Notices*, vol. 42, no. 6, pp. 156–166, 2007.

[8] A. E. Eichenberger, J. K. O'Brien, K. M. O'Brien, P. Wu, T. Chen, P. H. Oden, D. A. Prener, J. C. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, M. K. Gschwind, R. Archambault, Y. Gao, and R. Koo, "Using Advanced Compiler Technology to Exploit the Performance of the Cell Broadband EngineTM Architecture," *IBM Systems Journal*, vol. 45, no. 1, pp. 59–84, 2006.

[9] P. Bellens, J. M. Perez, R. M. Badia, and J. Labarta, "CellSs: a Programming Model for the Cell BE Architecture," *ACM/IEEE Super Computing Conference (SC'06)*, vol. 0, p. 5, 2006.

[10] D. Andrews, W. Peck, J. Agron, K. Preston, E. Komp, M. Finley, and R. Sass, "hthreads: A Hardware/Software Co-Designed Multithreaded RTOS Kernel," in *Proceedings of the 10th IEEE International Conference on Emerging Technologies and Factory Automation*, September 19-22, 2005.

[11] J. Agron, W. Peck, E. Anderson, D. Andrews, E. Komp, R. Sass, F. Baijot, and J. Stevens, "Run-Time Services for Hybrid CPU/FPGA Systems On Chip," in *Proceedings of the 27th IEEE International Real-Time Systems Symposium (RTSS)*, December 2006.

[12] D. Andrews, R. Sass, E. Anderson, J. Agron, W. Peck, J. Stevens, F. Baijot, and E. Komp, "Achieving Programming Model Abstractions For Reconfigurable Computing," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 1, pp. 34–44, January 2008.

[13] J. Adomat, J. Furuns, L. Lindh, and J. Starner, "Real-Time Kernel in Hardware RTU: A Step Towards Deterministic and High-Performance Real-Time Systems," in *Proceedings of the 8th Euromicro Workshop on Real-Time Systems*, 1996. [Online]. Available: citeseer.ist.psu.edu/adomat96realtime.html

[14] V. J. M. III and D. M. Blough, "A Hardware-Software Real-Time Operating System Framework for SOCs," *IEEE Design & Test*, vol. 19, no. 6, pp. 44–51, Nov/Dec 2002.

[15] J. Lee, V. Mooney, K. Instrom, A. Daleby, T. Klevin, and L. Lindh, "A Comparison of the RTU Hardware RTOS with a Hardware/Software RTOS," in *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC)*, Kitaskyushu International Conference Center, Japan, Jan. 2003, pp. 683–688.

[16] B. Senouci, A. Kouadri M, F. Rousseau, and F. Petrot, "Multi-CPU/FPGA Platform Based Heterogeneous Multiprocessor Prototyping: New Challenges for Embedded Software Designers," *The 19th IEEE/IFIP International Symposium on Rapid System Prototyping, 2008. RSP '08*, pp. 41–47, June 2008.