

Architectural Frameworks for MPP Systems on a Chip

David Andrews, Douglas Niehaus

Information Technology and Telecommunications Center (ITTC)

University of Kansas

{dandrews,niehaus}@ittc.ukans.edu

Abstract

Advances in fabrication techniques are now enabling new hybrid CPU/FPGA computing resources to be integrated onto a single chip. While these new hybrids promise significant performance increases through the customization of massive gate level parallelism, their full potential will not be reached until a suitable computational framework has been developed. We believe that the computational framework should provide a unified model that brings the FPGA and CPU based components under a common programming model for MPP developers. In this paper, we discuss extending the thread programming model to support hybrid CPU and FPGA computational components to allow systems programmers to access the MPP level of parallelism potential of the FPGA, but within a familiar and understood programming model.

1. Introduction

Massively Parallel processing (MPP) computations involve a large number of concurrent components with varying communication and execution constraint relationships. While no single programming model is appropriate to describe every MPP application, the familiar multi-threaded model is popular for many applications having components sharing both a machine and an address space. However, while support for multi-threading is obviously desirable, it is not sufficient for a number of reasons, including MPP application scale, fine grain resource use, and need for specialized fine-grain resource management from the supporting system software. Many large-scale MPP applications are implemented as multiple processes on multiple physical machines connected by a network. The use of Networks of Workstations (NoW) is thus attractive from a price/performance perspective, but extracting close to the best possible performance from a NoW is a difficult task.

Techniques in several aspects of hardware architecture and fabrication have improved to the point where a System-on-a-Chip (SoC) can contain a power general purpose CPU, substantial reconfigurable logic (Field Programmable Gate Arrays (FPGA)), memory

resources accessible to both CPU and FPGA, and a number of I/O channels supporting both communication within the SoC as well as with external devices. Small single board computers are thus now emerging with a powerful and unique mix of capabilities. At the system software level, Linux has improved significantly in both quality and power over the last several years, while concurrent advances in real-time extensions to Linux have provided a number of ways to exercise fine-grain resource control, and to support a number of different programming and scheduling models. Recent advances in FPGA programming techniques have demonstrated that higher-level programming language constructs can be used to specify gate-level behavior within FPGAs, thus substantially expanding the set of developers who can use them to support computations beyond those comfortable with the vocabulary of gate-level logic. Greater ease of use and increases in price/performance ratio of FPGAs has made it possible to create specialized FPGA support for specific computations components, increasing their execution speed by moving them into hardware, as well as *substantially increasing the parallelism* of the resulting hardware support for the application. Hybrid computations combining FPGA and CPU based components hold substantial promise for improving MPP application performance by decreasing the execution time of components moved into the FPGAs, as well as by potentially increasing the parallelism of the computations.

We believe that the realization of a computational framework unifying the FPGA and CPU based components under a common programming model is crucial for these powerful new hybrid systems to meet the needs of MPP developers and applications. This presents a fascinating design challenge as the hybrid system software must provide fairly general support to address a wide range of applications, but must also be capable of providing the specialized support required to substantially improve a specific application's performance. This is a difficult challenge in part because it requires the simultaneous satisfaction of apparently contradictory forces: generalization and specialization. This paper outlines a system

architecture that we believe is capable of resolving these seemingly contradictory requirements. To meet this challenge we are combining innovations in operating system design, implementation of computations within FPGAs, and significant extension of the range of programming models available to the application developer.

At the application level we will use the familiar and portable multi-threaded programming model to enable the implementation of hybrid MPP application software. Our approach is to enable well-understood synchronization and control methods for threads to execute on both the CPU and FPGA hardware, and developing a co-design of the operating system with the application thread-level libraries they support. At the operating system level we have several goals, including: fine-grain resource control, minimizing ERL, and configurable support for a number of different programming models, including user-specified scheduling policies. For the last five years, one of the authors has been developing the KU Real-Time (KURT) extensions to Linux [17], which already provides some of the desired support, and provides an excellent foundation within which to address the missing pieces. KURT-Linux already provided a number of ways in which to exercise fine-grain resource control, and we are currently finishing support for multiple programming models, including user-specified scheduling decision functions. We have also recently demonstrated that a modification to interrupt processing semantics can reduce ERL to less than 10 microseconds. Fully realizing this capability will require an application of the CPU/FPGA co-design techniques we want to apply to MPP applications. We are currently developing hybrid designs, those containing both CPU and FPGA based components, of several KURT-Linux components to help improve overall performance as well as generalized support for hybrid application-level computations.

We believe that hybrid designs will play an important role in improving MPP application performance for three major reasons. First, because hybrid designs will be required to produce support at the operating system level that has the necessary properties for MPP applications. Second, because FPGAs will be invaluable for increasing the performance of specific components of MPP applications. Third, because for the foreseeable future it will be impractical to move an entire MPP application into the FPGA, thus requiring a CPU/FPGA hybrid implementation. The most obvious reason is the size of the FPGA, but even arbitrarily large FPGAs cannot host every possible MPP application, as some application components

will remain better suited for implementation on a general purpose CPU than in an FPGA.

Current hybrid computational models are still immature, generally treating FPGAs as computational accelerators that are invoked passively as subroutines, or for essentially independent portions of a data-flow computation requiring only input and output queues. Our approach expands this set of methods to fully support the more general computational demands of next generation parallel applications. Within our model, applications are realized as sets of threads distributed flexibly across the system CPU and FPGA assets and controlled using standard concurrency control mechanisms (e.g. mutex semaphores, counting semaphores, read-write locks, condition variables). This will allow programmers to coordinate the execution of all CPU and FPGA threads uniformly, to support the computation as a whole. This approach will provide the following significant advantages. First, the use of the familiar multithreaded programming model will greatly accelerate program parallelism at the highest level of abstraction. Second, whether the threads implementing a computation are CPU-based or FPGA-based will become just one more of the available design and implementation parameters under developer control, albeit one with resource use and performance implications.

2. Related Work

Many researchers are investigating new design environments and tools for supporting embedded processing and more specifically, for supporting programming reconfigurable computing devices. Projects such as Ptolemy [1], Khoros [2], MatLab [3], Handel C [4] offer high-level simulation environments. Others focus on low level technology and platform-specific support issues [5]. The Streams-C [6] project at Los Alamos Laboratories lies at a point between these extremes, introducing new language constructs and creating hardware/software libraries supporting stream oriented processing on FPGA devices. The project has targeted the Wildforce FPGA [7] boards as the initial platform, and has successfully implemented a small library of Streams-C FPGA-based functions callable from C programs. Their Malleable Architecture Generator (MARGE) tool set, which includes the Stream-C compiler, contains synthesis components that can generate application-specific hardware implementations from library function descriptions in Streams-C [8]. They have used their approach to generate SIMD architecture within the FPGA.

The Streams-C project has demonstrated the possibility of raising the level of abstraction at which FPGAs are programmed from one of gate-level parallelism, to that of threads of execution. The Stream-C compiler can also produce a multi-threaded software simulation of the FPGA-based computations. The latter is particularly significant, since it emphasizes that a multi-threaded software architecture is appropriate for describing these types of computations. Our crucial observation is that application implementation will use a *hybrid multi-threaded model* in many cases because application components will be implemented by both CPU and FPGA based threads.

2.1 Multi-threading

Multi-threading was introduced within the OS several decades ago to make better utilization of CPU resources in the presence of slower I/O devices, by switching out a thread that is waiting on a slow device and switching in a thread that is ready to run, thus better utilizing the CPU. More recently, the basic concepts of multi-threading have been adopted as an *explicit* programming model to support application level concurrency, and as an *implicit* computational technique for increasing throughput by hiding latencies. Our goal is to extend the multi-thread programming paradigm to enable developers to include hybrid computations containing both CPU and FPGA based threads. The literature on various aspects of multi-threaded programming is vast and cannot be covered completely within this section. Therefore, we discuss selected representative relevant work.

2.1.1 Thread based latency hiding

Latency hiding multi-threading techniques can be categorized as either fine-grained, or coarse-grained. Fine-grained multithreading machine architectures concentrate on efficient switching among threads at the instruction level, whereas coarse-grained machines concentrate on thread switching at a coarser temporal level corresponding to comparatively costly stalls in thread execution such as cache misses. Monsoon [9], Tera [10], and IBM RS64 III systems attempted to hide memory latencies through interleaving. For example, the Tera system interleaved threads to reduce the effects of stalls due to slower memory subsystems. The MIT Alewife machine [11] was a fine-grained multithreading architecture supporting distributed shared memory, and provided fast low-level synchronization support for shared memory as well as message passing paradigms. The Star-T [12] and Star-T Voyager [13] projects explored the use of multithreading, and combining customized and commodity approaches to building scalable

processors. A related effort is the Stanford FLASH [14] project, which incorporated a programmable coprocessor (Memory and General Interface Controller: MAGIC) for implementing coherence and synchronization primitives.

These efforts, as well as many others, illustrate a number of potentially useful low-level implementation strategies, but are complementary to our work because they concentrate on switching and concurrency control among threads on the *same processor*. Our work will, in contrast, concentrate on integrating FPGA-based threads by permitting supporting interaction among a hybrid set of threads *across* the FPGA-CPU boundary. We thus concentrate on creating a *programming and execution environment* that strongly integrates the FPGA-based and CPU-based application components under the familiar multi-threaded programming paradigm. This work makes few assumptions about the architecture of the CPU involved, and could use any of these machine architectures.

2.1.2 Multi-Threaded Programming Paradigm

The multi-threaded programming model has gained acceptance as a standard way to implement application level concurrency. The model provides us with a strong set of operations controlling concurrent computations including hybrid computations with both CPU and FPGA based components. The general acceptance of the multi-threaded programming model led to the definition of a platform independent operating system standard (POSIX) [15]. The POSIX standard is fairly exhaustive, and addressing each and every element of the API is not necessary for our research. In this respect, the proposed work is not a fully transparent integration of support for FPGA-based threads under the POSIX model, but we do not believe it should be. The nature of the proposed support will be discussed in the Approach Section.

The corner stone of concurrency control, however, is hardware support for atomic operations on semaphore variables, and so CPUs for the last two decades have supported atomic operations such as test-and-set, fetch-and-add, or compare-and-swap [16]. The CPU supported instructions provide the basis upon which programs or middleware have built higher-level operations such as mutual exclusion, counting semaphores, critical regions, and conditional variables. Our work will extend support for synchronization and other relevant thread operations to hybrid sets spanning the CPU-FPGA boundary. A key aspect of this will be providing consistent support for the atomic test-and-set operation from FPGA

based computations to semaphores in main memory, and from CPU based computations to semaphores in the FPGA.

2.2 Event Response Time Considerations

Kurt-Linux has developed methods for handling real-time concerns within Linux, but RTLinux, and a related project RTAI [19], support real-time computations with a separate executive, running Linux as a non-real-time best-effort computation. Other projects have taken a similar approach to KURT-Linux by addressing real-time within Linux, including RED-Linux [20], which emphasizes selectability of scheduling models and preemptability, and Linux/RT [21], which emphasizes a share-based resource scheduling model. Another project has produced the Linux Kernel Preemption Patch, decreasing the worst-case latency for preempting a process, and has recently been incorporated into the standard Linux (2.5) source tree [22,23]. This approach has improved event response latency (ERL), but has not addressed the major remaining sources of ERL and inaccuracies in real-time scheduling: hardware interrupts blocking and interrupt handlers. Recent work by the investigators for this proposal have recently extended KURT-Linux to include interrupt handling in the programming model explicitly, and have shown that this can eliminate the majority of remaining ERL and scheduling jitter for real-time processes. The use of KURT-Linux will ensure that the hybrid computation support methods will be freely distributable, and fully visible to all interested parties; including commercial vendors such as Wind River (VxWorks) [24].

3. Approach

To support our objectives, we are creating a programming and execution environment that strongly integrates hybrid CPU/FPGA-based computational components under the familiar multi-threaded programming paradigm. Our successful outcome will be a fluid HW/SW co-design and execution environment within which applications can be realized as a set of computation threads with many possible mappings of threads to the CPU and FPGA processing assets. Our approach is to bring both FPGA based and CPU based components of an application under the umbrella of the POSIX PThreads model. This approach first requires enabling the FPGA to support atomic test-and-set operations, and then performing two separate but highly related co-design efforts. Providing atomic test-and-set operations is fundamental for supporting semaphore operations, which in turn are used for thread synchronization.

The first co-design effort is focused on developing appropriate low-level functionality across and between the CPU/FPGA assets within the operating system. The key issue in this co-design effort is to address how to decompose and migrate operating system functionality across the CPU/FPGA boundary. Proper co-design of the operating system features involved in the co-design effort will improve predictability, performance, and provide needed support for our hybrid computation model. The second co-design effort is at the application level to provide uniform concurrency and control for CPU and FPGA based threads by modifying and expanding the POSIX multi-threaded library. Each of these efforts is discussed below.

3.1 Enabling Atomic Operations

Although implementing semaphore semantics from within the FPGA appears straight-forward, design tradeoffs are required to determine where best to implement each semaphore variable, and what hardware support should be available on a per thread basis for performing atomic operations on the variable. Although multiple CPU threads appear to execute concurrently, they are still executed sequentially on the CPU, which is often termed pseudo-concurrency. The concurrent threads running within the FPGA can truly run in parallel, often termed physical concurrency. If each FPGA thread executes independently then each must have an associated bus request capability. This has the potential of introducing long delays and blocking. A second alternative is to develop an FPGA based local arbiter to coordinate FPGA component thread bus requests. In this case, an additional hardware model would be required to field the multiple bus requests, and then generate a request to the system bus arbiter. The bus request device must be able to queue, sort, and handshake with each requesting FPGA thread. In either approach, protocols and supporting signal interfaces will need to be defined. The research issue is to determine the best design approach.

We place no restrictions on requiring semaphores to be implemented in DRAM memory and, in fact, our approach allows the variables to be implemented within the FPGA. Certain advantages are gained under this approach, including reducing system bus traffic, as the multiple threads that are resident within the FPGA do not have to individually arbitrate for the bus. Additional performance advantages will result from not having accesses slowed by multiple bus cycles and wait states associated with typical DRAM memory. These performance advantages should be evaluated against the additional complexity and gate

counts associated with memory address decode and control logic within the FPGA. The fluidity of placing the semaphore variables in either the DRAM or FPGA is an asset of our approach and systems may contain both implementations at any given time.

3.2 Operating System Co-Design

A central research challenge is defining a new partitioning of operating system services across the hardware/software boundary. The FPGA threads cannot request system services using the same mechanisms as CPU based threads, such as traps to the operating system. Threads running within the FPGAs also contain different state information when compared to CPU resident threads, requiring corresponding modifications to the co-designed system software.

Our approach is to preserve essentially the same semantics as existing implementations of system *services*, but in a new CPU/FPGA partitioning that exploits the opportunities provided by the FPGA devices. Of particular interest is our goal of reducing the overhead that currently is associated with performing specific real-time system services on the CPU. This overhead eliminated by migrating to the FPGA results from both the processing time the system software spends in executing specific service handlers on the CPU, but also any associated context switching times.

In addition to reducing the overhead, our approach will also provide several enhanced capabilities for embedded systems. We will investigate enhancing the resolution and decreasing the jitter in scheduling. We will achieve this by pursuing our current efforts in moving KURT-Linux time keeping and scheduling into the FPGA. Second, we will investigate adopting a new approach we have developed for interrupt processing. In our current effort we have defined a new hardware/software partitioning of interrupt processing that shows a decrease in the overhead for evaluating and scheduling external interrupt requests. We will start our investigation by evaluating the partitioning of these system services that are critical to embedded systems. It is our belief that migrating portions of these functions into the FPGA will provide significant performance gains. We outline our investigations of these functions below.

3.2.1 Semaphores

Mutual exclusion uses a binary semaphore to control the sequence in which multiple threads can access and manipulate data. Threads running on the FPGA must

be able to achieve these semantics for guaranteeing mutually exclusive access to data shared with CPU based threads. Test-and-set operations can be provided by the underlying hardware. The remaining semantics will require hardware software co-design of the system software and control functionality embedded within the FPGA. The conditional test can easily be placed in hardware with a simple comparator causing the controlling state machine to transition into an idle state and generating an exception request to the scheduler. The exception request can be an external interrupt if the scheduler is running on the CPU, or as discussed in the thread control section, may be an exception to the scheduler implemented within the FPGA device itself. In either case, the scheduler must record the thread ID, and place the thread on the suspend queue.

For CPU based threads, the system software restarts threads by switching context to the thread that is ready to run and has been selected by the scheduler. All CPU threads waiting for a particular semaphore are marked as ready by the *wakeup* facility when the semaphore is released. For threads running on the FPGA, a new approach will be required to restart threads depending on the sophistication of the hardware implementation. As an example, system software can restart the thread by writing into a memory mapped register that has the effect of moving the state machine from idle back into the execute state. Additional advantages are gained by implementing the semaphore in a memory-mapped register within the FPGA.

First, the semantics do not change, only the physical device in which the memory mapped variable resides. However, significant savings in bus activities can be achieved as all threads that run within the FPGA can access the variable without requiring a bus request. The atomic nature of the operation between CPU and FPGA resident threads will still be guaranteed if the FPGA monitors the address requests from the CPU across the bus. If the FPGA decodes a request to a semaphore variable, then it can internally block all requests to the variable while the CPU thread completes the test-and-set. If the CPU requests access to a variable that is currently being evaluated by a FPGA thread, then the request can be delayed until the FPGA thread has completed. Additional savings can be gained in the time required to perform the operation for hardware threads as the arbitration for simultaneous accesses to the variable does not require arbitrating for the bus, and waiting for slower main memory.

The implementation tradeoffs for supporting counting semaphores within the FPGA are similar to those associated with the binary semaphore. Barrier synchronization is also fundamental to the thread model. Supporting barrier synchronization on CPU based threads typically requires first implementing a binary semaphore that protects the counter. A second mutex operation is typically performed on a variable that initiates execution of all threads. These semantics can be achieved with the FPGA threads by implementing the mutex operation as discussed above. In addition to these basic semaphore operations, other synchronization primitives, such as read/write locks, have been defined in other related thread packages. After the basic semaphore techniques have been developed, these other primitives can easily be implemented.

3.2.2 Scheduling

KURT-Linux currently provides two orders of magnitude finer temporal resolution for event scheduling than conventional Linux. KURT-Linux increased the time keeping resolution by using the Time Stamp Counter (TSC) on the x86 architecture that ticks at the CPU clock rate as the “time standard”. We increased resolution of event scheduling by using the microsecond resolution timer chip in the standard PC architecture to schedule the “next” event. KURT-Linux preserves conventional Linux’s tick counter, the 10ms “jiffy” for compatibility, keeping track at a finer resolution in a separate “sub-jiffy” variable. The sub-jiffy resolution is architecture dependent due to CPU clock rate differences, as well as variations in time standard and event scheduling support. However, KURT-Linux is quite portable, and currently runs on both the StrongArm and XScale processors as well as the x86. KURT-Linux supports time standard calibration for each CPU to determine the number of sub-jiffy counter ticks that comprise a standard jiffy. When a real-time event timer interrupt occurs, the scheduler is invoked to switch control to the real-time process associated with the event, calculate the time to the next real-time event for the system, and then reprogram the event interrupt device.

We are currently migrating these functions into the FPGA device, which will provide the following gains. First, the sub-jiffy resolution can be selected as part of the system design. The resolution of the sub-jiffy can be set (up to a minimum that depends on the FPGA clock frequency) within the FPGA as a function of the register size. This will permit application designers to balance event-scheduling resolution with FPGA resource consumption. Second, the calculations that are required to determine the next timer event can be

computed in hardware, thus eliminating the overhead associated with each timer event. Third, a portion of the scheduling algorithm can also be placed into the FPGA, further decreasing event handling overhead, the associated event response latency (ERL) and, more subtly, the ERL *variance*. This is particularly important because reducing ERL variance helps improve the *predictability* of system support for MPP applications. Finally, we will shift the majority of the exception processing for these functions into the FPGA. This will reduce the number of context switches that occur, further improving the predictable execution of CPU based MPP computations, even without direct FPGA support for MPP computation components.

3.2.3 Interrupt and exception processing

We have been working on modifying the interrupt processing semantics in KURT-Linux for several months, and have succeeded in reducing ERL at the OS level below 10 microseconds with variance of only 1-2 microseconds for 10^7 events. We have just completed an initial experiment with 10^5 events showing similar results for ERL in transferring control to a user process. There are still problems associated with heavy disk load for the user process experiments, so the results are preliminary but promising. However, the method we are using depends on detailed consideration of current semaphore use, the execution interleaving constraints this imposes on the current set of event handlers, and execution of an interrupt handler decision function. The efficiency of the constraint analysis and decision function has a crucial influence on the success of our approach, and will thus be an excellent candidate for analysis of ways in which FPGA support could be beneficial. A particularly interesting possibility is to completely migrate scheduling of exception processing into the FPGA, since much of the processing first done when entering an interrupt service routine is to determine what device or event caused the interrupt, and under the new KURT-Linux approach, to decide if the handler should be executed immediately or deferred. This functionality can be realized by several approaches within the FPGA. Moving this processing into the FPGA, along with scheduling and the timers will significantly reduce the number of asynchronous events that the system software must evaluate on the CPU. This is of tremendous potential significance because it is likely to produce a major improvement in the predictability with which MPP computations are executed. We are particularly hopeful and confident that this is likely because of one of the investigator’s experience as one of the principle architects and developers of the Spring real-time system [26], which

demonstrated that segregation of the scheduling and real-time application computations produced a significantly more predictable system. Further, experience with the Spring Scheduling Co-Processor [27,28] will be extremely relevant and helpful in the HW/SW co-design process choosing how to distribute aspects of the system functions across the CPU/FPGA boundary.

3.3 Application Level Co-Design

The acceptance of the multi-threaded programming model has led to the definition of a platform independent operating system standard (POSIX) [15] that supports application level multi-threaded programming. The POSIX standard is fairly exhaustive and addressing each and every element of the API is not necessary for our research. In this respect, the proposed work is not a fully transparent set of support for FPGA-based threads under the POSIX model, nor do we believe it should be. The *concurrency control* portion of the POSIX model should be fully transparent, including its implications for thread scheduling at the OS level. However, we believe other aspects of the model will be best handled by *adding* analogous API elements for FPGA control, rather than *encapsulating* within the existing POSIX routines. For example, our first version of FPGA-thread creation will be handled by an additional routine as the management of FPGA resources is sufficiently different semantically that encapsulation could introduce as much confusion as advantage. We will approach an API iteratively, first implementing basic FPGA-thread creation and deletion routines, and then concentrating on concurrency control in detail. After we have fully explored the concurrency control aspects of the API, we will consider less central features and reevaluate whether encapsulating FPGA-thread creation and deletion within the existing POSIX API is desirable. Some of the issues that we address in our application level co-design are discussed below.

3.3.1 Thread Control

FPGAs require that the CLBs (combinational logic blocks) be programmed prior to execution. The algorithm that has been specified for a thread must be configured into the FPGA prior to run time. In effect, the operations and interconnections that form data paths and computations for the hardware computational components must be pre-loaded without allowing the FPGA based thread to execute. To invoke and suspend the thread will require the creation of new finite state machine (FSM) constructs that will perform the thread control duties for FPGA

based threads that *traps* perform for CPU based threads. When a thread is loaded into the FPGA, it should contain control states that represent initialization, execution, suspension, and termination.

Upon initial configuration, a thread should default to the initialization state and require invocation from the operating system to start. This invocation can be achieved by writing into a memory mapped control register that is used by the `create_thread` API. Once the control word has been written, then the FSM transitions into the run state. The FPGA thread continues execution until it reaches either a synchronization point or termination. When a synchronization point is reached, any further execution must wait until the result of the synchronization operation permits progress. In the case of a *wait* operation (P) on a semaphore, this may imply that the thread must spin, or be suspended in a fashion semantically equivalent to a CPU thread blocking. If the thread is suspended, the FSM must transition into an idle state and wait for a proper restart signal. The realization of the restart signal will depend on the hardware/software partitioning of the system software, and on how the FSMs suspended state is represented to the OS. When the FPGA thread is to be re-started, the operating system can write a control word into the thread's associated memory mapped control that invokes the state machine to transition from the suspend to the execute state.

We can thus provide the basic foundation for control required to coordinate a hybrid set of CPU and FPGA based threads implementing an application. Higher-level control semantics will be addressed by extending existing POSIX library synchronization routines to include methods for handling hybrid computations. We will investigate the utility of these approaches using both abstract multi-threaded code test cases, and within the context of multi-threaded audio applications for Linux that provide excellent driving examples of computations with requirements essentially identical to those of more obvious RTEC computations such as process or machine control. The advantage of the audio examples is that they are readily available, popular, and use inexpensive COTS hardware for playback.

Bibliography

- [1] Lee, Edward, "Overview of the Ptolemy Project", Technical Memorandum, UCB/ERL MO1/11, University of California, March 6, 2001
- [2] www.koral.com
- [3] www.mathworks.com

- [4] www.iis.ee.ic.ac.uk/~frank/surp99/article1/ama_g97/
- [5] Xilinx
<http://www.xilinx.com/xilinxonline/jbits.htm>
1999
- [6] Maya B. Gokhale, Janice M. Stone, Jeff Arnold, and Mirek Kalinowski, "Stream-Oriented FPGA Computing in the Streams-C High Level Language", Proceedings of the Eighth Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), B. L. Hutchings, ed., Napa, CA, April 2000, 49-56.
- [7] www.annamicro.com
- [8] Maya Gokhale, James Kaba, Aaron Marks, and Jang Kim, "A Malleable Architecture Generator for FPGA Computing", Proceedings of the SPIE - The International Society for Optical Engineering, 1996, vol. 2914, 208-217.
- [9] Papadopoulos, G., and Culler, D., "Monsoon: An Explicit Token-Store Architecture", Proceedings of the 17th Annula International Symposium on Computer Architecture, pp. 82-91, June 1990
- [10] Alverson, G., Alverson, R., and Callahan, D., "Exploiting Hterogeneous Parallelism on a Multithreaded Multiprocessor", Proceedings of the Workshop on Multithreaded Computers, Supercomputing 91, November 1991
- [11] Agarwal, A., Bianchini, R., Chaiken, D. Johnson, K., Kranz, D., "The MIT Alewife machine: Architecture and performance", Proceedings of the International Symposium on Computer Architecture, Denver Co., June 2-13, 1995
- [12] Nikhil, R., Papadopolous, G., and Arvind,"*T: A multithreaded massively parallel architecture", Proceedings of the 19th International Symposium on Computer Architecture, Gold Coast, Australia, May 1992, pp. 156-167
- [13] Ang, B., Chiou, D., Rosenband, D., Ehrilich, M., Rudolph, L., and Arvind,"StarT Voyager: A flexible platform for exploring scalable SMP issues", Proceedings of Supercomputing 98, Orlando, Fla, Nov. 1998
- [14] Kuskin, J., Ofelt, D., Heinrich, M., Heinlein, J., Simoni, R., Gharachorloo, K., Chapin, J., Nakahira, D., Baxter, J. Horowitz, M., Gupta, A., Rosenblum, M., and Hennessy, J.L., "The Stanford FLASH multiprocessor", Proceedings of the 21st International Symposium on Computer Architecture, Chicago. IL., April 1994
- [15] <http://www.opengroup.org/onlinepubs/oo7904975/toc.htm>
- [16] John P. Shen, Mikko H. Lipasti, "Modern Processor Design", McGraw Hill Beta Edition, 2002
- [17] B. Srinivasan, S. Pather, R. Hill, F. Ansari, D. Niehaus. "A Firm Real-Time System Implementation Using Commercial Off-The Shelf Hardware and Free Software", *Proceedings of the Real-Time Technology and Applications Symposium*, Denver, June 1998.
- [18] FSM Labs, "RTLinux", <http://www.fsmlabs.com>
- [19] DIAPM RTAI, <http://www.aero.polimi.it/~rtai/>
- [20] Y. Wang and K. Lin, "Implementing a General Real-Time Scheduling Framework in the {RED}-Linux Real-Time Kernel", IEEE Real-Time Systems Symposium, 1999, pp. 246-255. url: citeseer.nj.nec.com/wang99implementing.html
- [21] TimeSys. "TimeSys Linux", <http://www.timesys.com>
- [22] Linux Kernel Preemption Patch Home Page, <http://www.tech9.net/rml/linux/>
- [23] G. Anzinger and N. Gamble, "Design of a Fully Preemptable Linux Kernel", <http://www.linuxdevices.com/articles/AT4185744181.html>
- [24] VxWorks, <http://www.windriver.com>
- [25] www-3.ibm.com/chips/roducts/coreconnect/
- [26] J. Stankovic, K. Ramamritham, D. Niehaus, M. Humphrey, "The Spring System: Integrated Support for Complex Real-Time Systems," *Special Issue of Real-Time Systems Journal*, Vol 16, No. 2/3, May 1999.
- [27] W. Burleson, J. Ko, D. Niehaus, K. Ramamritham, J. Stankovic, G. Wallace, C. Weems, "The Spring Scheduling Co-Processor: A Scheduling Accelerator," *IEEE Transactions on VLSI Systems*, Vol 7 No 1, March 1999, pp. 38-48.
- [28] D. Niehaus, K. Ramamritham, J. Stankovic, G. Wallace, C. Weems, W. Burleson, J. Ko, "The Spring Scheduling Co-Processor: Design, Use, and Performance", *Proceedings of the IEEE Real-Time Systems Symposium*, Raleigh-Durham, NC, December 1993.