Building Blocks for a New Generation of Application-Specific Computing Systems

Brent Baxter, George Cox, Thomas Gross, H. T. Kung, David O'Hallaron, Craig Peterson, Jon Webb, and Paul Wiley

School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

Intel Corporation, CO4-01
5200 N.E. Elam Young Pkwy
Hillsboro, Oregon 97124

Abstract

Carnegie Mellon University and Intel Corporation have jointly developed the iWarp processor, which integrates both communication and computation functions on a single VLSI component. The iWarp component and subsystems including it are powerful building blocks for constructing a new generation of application-specific computing systems. These special-purpose systems can achieve very high-performance, while maintaining a high degree of flexibility to address different needs of an application. In particular, iWarp systems deliver high computation bandwidth (up to 20 GFLOPS for a 1024 cell system), as well as high communication bandwidth (320 MBytes/sec per cell). And programming these systems is assisted by modern tools such as optimizing compilers and parallel program generators.

1. Introduction

General-purpose computers often cannot meet the requirements of the most demanding applications, such as radar and acoustic signal processing and high-performance image processing. These applications have high computing requirements, which are typically on the order of several GFLOPS (billion floating-point operations per seconds). In addition, systems for for these application are usually required to have low energy consumption and be compact in their physical sizes. Only *application-specific computing systems*, whose designs are tailored to the application, can meet these requirements. These systems can typically make use of large-scale parallelism with hundreds or thousands of powerful processors.

By application-specific computing systems we mean systems that cover complete *application systems* (e.g., over-the-horizon radar), not just individual application *tasks* (e.g., controlling a disk or printer) that are often implemented by gate arrays or ASICs. An entire application usually involves complex algorithms and floating-point arithmetic. Application-specific systems for these applications therefore need to use VLSI components such as programmable array logic and high-speed floating-point chips. These components give the designer complete freedom to

---

The research was supported in part by Defense Advanced Research Projects Agency (DOD) monitored by DARPA/CMO under Contract MDA972-90-C-0035, and in part by the Office of Naval Research under Contract N00014-90-J-1939.

Authors' affiliations: B. Baxter, G. Cox, C. Peterson, and P. Wiley are with Intel Corp., T. Gross, H. T. Kung, D. O'Hallaron, and J. Webb are with Carnegie Mellon University.

custom-tailor the hardware design, thereby achieving high performance for a given power or space budget. However, this approach has the following disadvantages:

1. Low level of integration. ‘‘Glue logic’’ is needed to build a parallel system out of standard components or processors.

2. High development cost. Each computing system is custom-designed, and the design is normally amortized over only a few systems.

Because of these drawbacks, the production costs of these systems are high.

In addition, the more specialized the hardware, the more difficult it is to adapt to algorithm improvements, future upgrades, wider operational dynamics, and changing system requirements. Fielded systems can be obsolete by the time they reach operational status. To overcome the inflexibility of the hardware, most systems allow some amount of programming to make possible future adjustments to changing requirements, as well as corrections to the original design. However, application-specific computing systems do not create a software base. If the processors are custom designed, there is no programmer community outside of the user group for that specific system, which implies that all software tools, even such basic tools as an assembler or compiler, must be newly developed. If the base processor is widely used the most basic tools can be shared, but other software components, such as a runtime system, must still be generated for each system.

Users have accepted these disadvantages because application-specific computing systems built from discrete components were the only way to meet performance requirements. However, the user requirements continue to grow. Specific systems begin to cover more complete applications: they require that the hardware executes more than just a kernel or the inner-most loops of an algorithm. This means that application-specific systems must deliver more computation bandwidth, more communication bandwidth, and more functionality. As the bandwidth demands continue to increase, application-specific computing system must reduce their cost to be affordable in the future. Not only are hardware costs important, even more critical is that a solution to the software cost problem must be found. Better software tools that provide a higher level of abstraction are needed, and these tools must be shared across multiple systems.

The iWarp component [2, 3], joinly developed by Intel and Carnegie Mellon, intends to solve some of the problems mentioned above. iWarp is unique in that it integrates computation, an instruction cache, a large register file, and communication all on a single VLSI component. By providing support for both computation and communication, iWarp is a suitable target for program development tools and forms the core of building blocks for high-performance parallel computing systems.

2. iWarp overview

The iWarp design is heavily influenced by our experience with the Warp machine [1] and has been described in more detail in earlier papers [2, 3]. In this section we briefly summarize the iWarp component and the iWarp system.

2.1. iWarp component

The iWarp component is a single-chip processor that requires only the addition of local memory to build a complete system building block called the iWarp cell. There are three distinct sub-components: the computation agent, which performs the normal, explicitly programmed computation; the communication agent, which handles input/output with adjacent cells, and the memory unit, which provides access to the memory for both the computation and the communication agent.

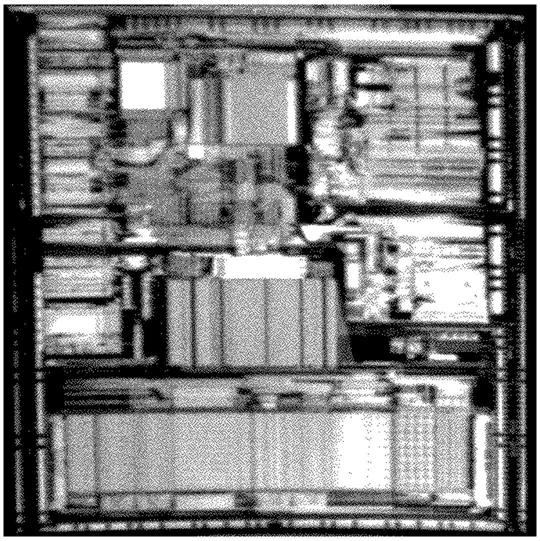

Figure 2-1: Photo of iWarp component

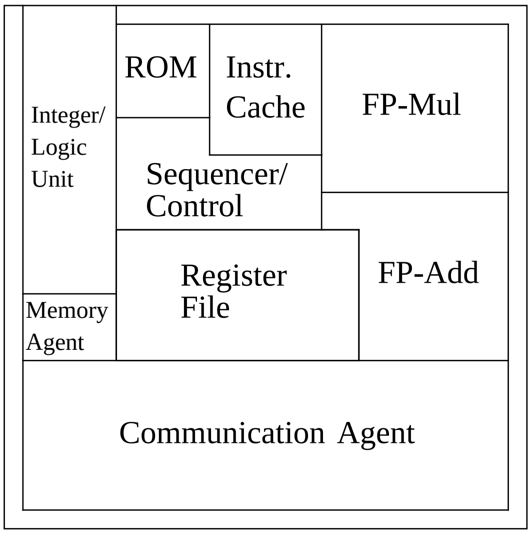

Figure 2-2: Floor plan of iWarp component

The computation agent includes a 20 MIPS integer/logical unit, hardware floating-point (20 MFLOPS single precision or 10 MFLOPS double precision, without pipelining), hardware support for divide, remainder, and square root, a multiported 128 word register file, a 256 word instruction cache RAM, and a 2K word ROM for built-in functions and constants. All of the functional units can operate in parallel using 96-bit wide instructions [4], or can be controlled independently using 32-bit wide instructions.

The communication agent connects to eight external physical busses with a total bandwidth of 320 MBytes/s. Multiple logical channels are multiplexed on each physical bus, with word-by-word, hardware flow control for each channel in such a manner that logical channels do not consume any physical bus bandwidth when idle. Each logical channel includes a hardware queue of depth 8 words. Up to twenty incoming and up to eighty outgoing logical channels are supported simultaneously by each iWarp component. The interface between the communication agent and the computation agent allows data to be read and written directly (without passing through external memory) via registers, called systolic gates, and supports blocking when reading from an empty or writing to a full queue.

The memory unit supports transfers between either the computation agent or the communication agent and external memory. There are separate address (24-bits wide) and data (64-bits wide) busses. Up to 20 million memory accesses/sec can occur, for a peak bandwidth of 160 MBytes/sec. The memory unit supports also the transparent spooling of data directly to and from the communication agent, in a DMA-like fashion.

2.2. Intercell communication

iWarp supports both systolic communication (individual data items are transferred as they are produced, or are used as they are received, by the program running at the processor) and memory communication (the entire message is buffered in the local memory of the processor before it is transmitted or after it is received) [3].

The iWarp communication agent includes hardware support for the dynamic set up of connections from one cell (source cell) to another cell (destination cell). Such connections, which may span two or more cells, are implemented by pathways [3]. A pathway is a sequence of logical channels on the intermediate cells and is long-lived; it stays up until taken down explicitly by the program executing on the source cell.

Long-range communication allow non-adjacent cells to communicate as if they were adjacent, with only a 200 ns/cell latency. This supports the implementation of occasional long-distance communication as well as the efficient simulation of communication patterns that do not map directly into the underlying hardware.

A long-lived connection allows the implementation of low-latency communication between cells. Once the connection is set up, no further overhead is associated with the transfer of data; programs on the source and destination cells can communicate on a word-by-word basis, which is essential for systolic communication.

Memory communication, which is used by well-known paradigms such as message passing, is supported by hardware message routing, as well as by spooling [3]. In addition, the logical channels can be divided into two pools, so that separate resources are used for memory and systolic communication. This guarantees that communication by the runtime system is never blocked by the communication of the user program.

2.3. Systems

iWarp systems are built out of iWarp cells. For a large number of applications, standard iWarp cells offer a cost-effective solution by providing a building block that forms the base for small as well as big arrays.

2.3.1. Standard boards

At this time, the standard iWarp board contains four cells; such a quad-cell board is shown in Figure 2-3. The board conforms to the double-height Euro-card form factor and provides between .5 and 6 MBytes of SRAM memory. Future plans may include ‘‘piggy-back’’ expansion boards that provide large memory expansion (e.g., 8 MBytes of DRAM) or direct I/O for a variety of peripheral interfaces. Larger memories will become available as denser memory chips appear, subject only to the limitation of 64 Mbytes/cell that is imposed by the number of address lines.

Figure 2-3: Photo of iWarp quad cell board

2.3.2. iWarp arrays

An iWarp array can have a small or large number of cells. Small arrays provide a cost-effective platform for program development. The small number of cells makes it feasible to place the array in the same cardcage that houses the CPU and memory of a typical workstation.

Large arrays, residing in a separate cage, can deliver high computation rates. The initial systems can be extended to include up to 1024 cells, delivering a peak performance of over 20 GFLOPS. Large systems can include disk systems for input/output, as well as one or more external connections for workstations, local-area networks, mass-storage devices, sensors, an HIPPI interface, etc., as depicted by Figure 2-4.

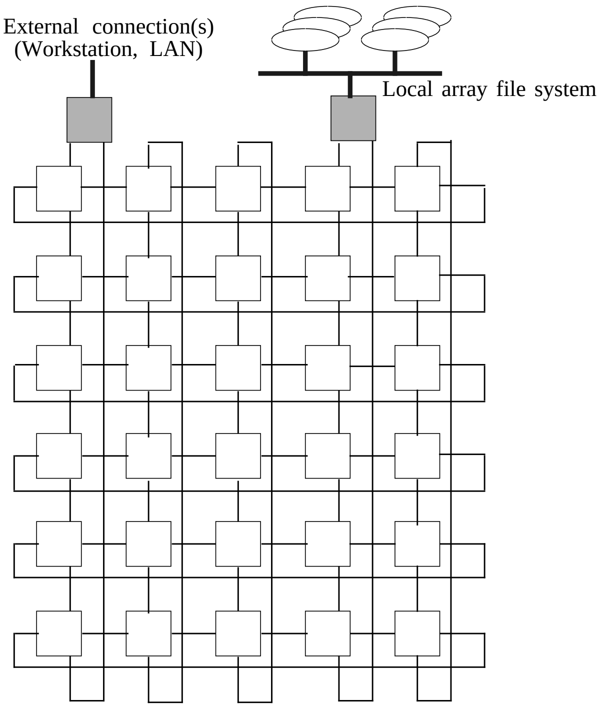

Figure 2-4: Standard iWarp system

2.3.3. Experimental systems

Each of iWarp's four input and four output busses operates independently. A cell can be connected to a neighbor cell by an input/output bus pair (as is the case for the standard boards described above), or the cell can be connected to a neighbor by an input bus only or an output bus only. This flexibility makes it possible to build new experimental systems with unconventional processor topology, which are too difficult or expensive to build with other components. Though this may be a minor use of the component as compared to the applications described above, it may support interesting research requiring new interconnection topologies.

For example, Figure 2-5 shows the use of the iWarp cell as the building block of a 3D array with or without wraparound connections. The eight undirectional ports of iWarp allow each cell in the 3D structure to have bidirectional communication along one dimension and unidirectional communication along the other two dimensions. iWarp can also be used to implement a hexagonal array, as shown in Figure 2-6. The integration of communication and floating-point computation on the same component makes iWarp an interesting building block component for novel switches such as fast combining switches capable of combining floating-point data.

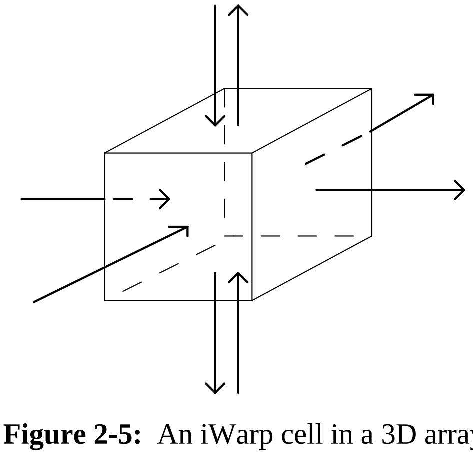

Figure 2-5: An iWarp cell in a 3D array

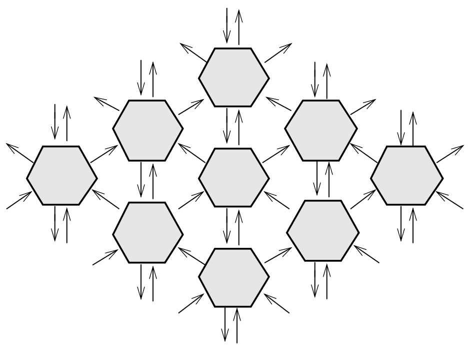

Figure 2-6: Hexagonal array

2.4. iWarp as a building block

The unique features of the iWarp component – pathways, flexible systolic communication, and efficient memory communication – make iWarp an ideal base for building blocks. iWarp can be used as a building block at multiple levels: the array level, the board level, and the component level. Each level leverages the key features in different way.

Logical channels are directly supported by the hardware and provide the foundation for connections between the programs executing on the cells. These connections, e.g., in an iWarp array like the one shown in Figure 2-4, can be configured to meet the requirements of an application. This will allow a number of application to avoid the expense of building a special system, since they can produce software to turn a generic array into an application-specific computing system.

Using iWarp boards, a variety of array configurations can be built. Using the backplane of the standard array, boards with different memory configurations can be combined to create triangular arrays, for example. With different backplanes, other shapes can be supported as well.

Finally, the iWarp component integrates a sophisticated communication agent with a powerful computation agent, which offers functionality and performance comparable to stand-alone micro-processors. Since these agents are on the same die, no additional glue logic is needed to join the computation engine to the communication system.

Basing these building blocks on the common iWarp component has the advantage that they all can share the same growth path. As the speed of the iWarp component increases over time, all parts of the building block family will benefit from the improvement.

3. Integrated support for application-specific systems

In this section we illustrate how the iWarp key features, pathways, systolic communication, and memory communication, provide a base for building special-purpose iWarp systems that are custom-tailored to specific applications. We use examples from the digital signal processing (DSP) domain; they range from a simple computational kernel to a complete application system.

Most DSP applications can be modeled as flow graphs, where graph nodes represent computations and graph edges represent communication channels. In an iWarp system, there is a natural correspondence between edges and pathways, and between nodes and cells. This mapping of edges to pathways and nodes to cells is the foundation for a simple approach to building special-purpose iWarp systems for a wide spectrum of applications. The pathways are set up by software at initialization time. Notice that arbitrary interconnection patterns can be created using pathways, and that once created, pathways impose no communication overhead. In the absence of contention, data are transferred at the maximum bandwidth of the physical busses. Notice also that the iWarp's systolic communication mechanism provides direct access to data on the pathways, as well as hardware flow control; a program on a cell simply reads or writes the appropriate systolic gate.

For example, Figure 3-1 shows a typical example of a computational kernel [5]. A matrix of sensor values is triangularized as part of an adaptive beamforming application.

The same mechanisms can be used to implement more complex computations. For example, Figure 3-2 shows a multifunction pipeline where filtering and thresholding kernels are chained together to form the front-end of a radar application. As before, pathways implement the interconnection pattern, and systolic communication is used to transfer data between cells.

Figure 3-3 shows a portion of the flow graph for processing sensor data from a sonobouy [9]. Each node in the graph corresponds to a *sequential* kernel computation (e.g., finite impulse response (FIR) filter or fast Fourier transform (FFT)). The behavior of each node is an infinite sequence of the form: (1) read inputs from input edges, (2) perform kernel computation, (3) write outputs to output edges. Again, each edge corresponds to a pathway. We could use systolic

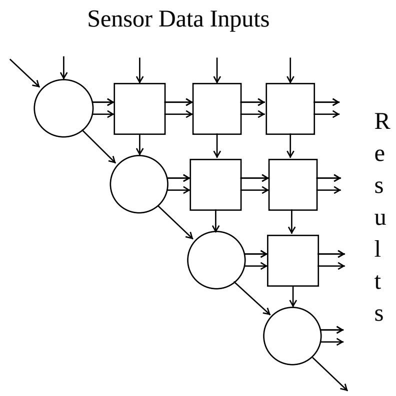

Figure 3-1: Triangular system

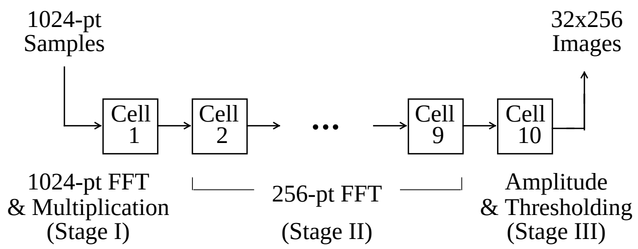

Figure 3-2: Multifunction pipeline system

communication for this example as we did in the previous examples. However, since each cell operates sequentially on large data sets, traffic will tend to be bursty and unpredictable. iWarp's memory communication mechanism is an attractive alternative for applications with this kind of large, bursty messages. Using spooling, it is easy to implement a double-buffering scheme that stores messages in memory as they arrive, without intervention from the computation agent. As with systolic communication, flow control is implemented in hardware.

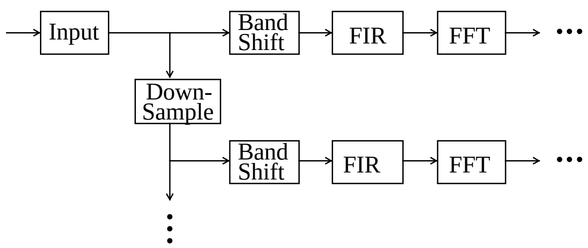

Figure 3-3: Heterogeneous signal flow system

The same mechanisms can also be used to build complete applications. For example, Figure 3-4 shows a system for processing sensor data from a collection of different sonobouys. (The processing for a particular type of sonobouy is called a *mode*, hence the name *multiple mode system*). A node corresponds to the computation for a particular sonobouy. An input edge corresponds to an input channel from a sonobouy. An output edge corresponds to an output channel connected to a display device. Again, pathways and the memory communication mechanism can be used to implement the application in a natural way.

To this point, we have assumed that applications are static, that is, their graphs do not change

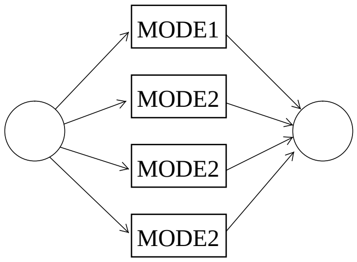

Figure 3-4: Multiple mode system

over time. However, many applications are dynamic, with graphs that change over time. For example, Figure 3-5 shows a multiple mode sonobouy system where the mix of modes and the number of instances of those modes changes over time. In this example, an instance of a new mode is created dynamically (i.e., a new bouy is dropped in the water), adding a new node and two new edges to the system. Since pathways can be constructed and dismantled dynamically, an iWarp-based system can be programmed to handle this kind of dynamic behavior, by allocating the node to an available cell, and by dynamically creating new pathways to and from that cell.

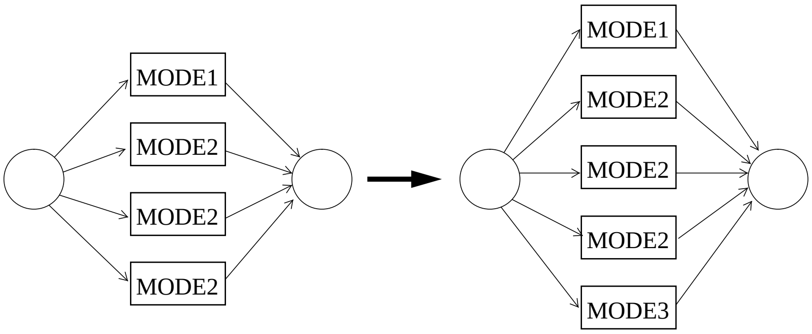

Figure 3-5: Dynamic multiple mode system

The previous four examples illustrate the central role of the pathway, systolic communication, and memory communication mechanisms in building special-purpose iWarp arrays for a wide spectrum of computations, from static computational kernels to large dynamic applications. In summary, the identification of cells with graph nodes and pathways with graph edges is a consistent and straightforward approach to building special-purpose arrays.

We anticipate that the pathway mechanism will also be useful in supporting fault tolerance and dynamic load balancing. Recent research has shown that pathways provide a workable basis for building fault tolerant systems [8]. The basic idea is that the interconnection pattern of an application can be preserved in the presence of faults by rerouting pathways dynamically, as faults occur. Similarly, it is not hard to imagine new dynamic load balancing schemes that exploit the dynamic nature of pathways.

4. Programming

There are two aspects to programming: the development, preparation, and maintenance of programs, and the execution of programs on the target system. The former requires a *tool chain*, the latter a suitable execution environment.

4.1. Parallel program generators

The programming of application-specific computing systems requires more than just a compiler or assembler for each processor in the system. Each parallel program incorporates a view of the parallel computer known as a *programming model*. The programming model gives the parallel computer certain effective performance characteristics and makes it capable of computing some kinds of operations efficiently, and others less efficiently or not at all. For example, in a signal processing program two programming models can be used – data partitioning and task queue. In data partitioning, each processor gets a portion of the data, and computes the output on its portion. This model can be used so long as the operation is local – for example, convolution. In the more general task queue model, the operation to be performed is divided into a large set of tasks, which are dealt out to processors, with each processor requesting a new task as it finishes an old one.

In a hand-written program the programmer builds up both the program and programming model. The programming model must be built from more general, and basic, constructs; as a result, the programmer sacrifices efficiency (since the general constructs are less efficient than special constructs especially for the programming model), suffers long system development time, and constructs a brittle program that is difficult to change (since the programming model constructed is specific to a program).

By contrast, with application- and programming model-specific development tools, the programmer writes only the program; the programming model has been built by someone expert in implementing that model on the parallel computer. Then this program is translated into programs for each cell in the system; these cell programs contain the low-level constructs required to implement the programming model. For this reason, we refer to these tools as *parallel program generators*. This approach provides the user with a higher level abstraction, which brings numerous benefits:

1. Clarity. A programmer specifies the problem at a higher level; instead of using the implementation level of the programming model, the level of the programming tool is used.

2. Simplicity. Application-specific tools can provide special programming constructs for the applications area and programming model, making the resulting programs more succinct and easier to write. For example, a tool for image processing can contain the concept of image boundary.

3. Efficiency. Application-specific tools can exploit expert knowledge about the applications domain and parallel computer to generate better programs than would normally be written by hand.

4. Flexibility and portability. Application-specific program generators generally hide the processor configuration from the programmer. The configuration can be changed without modification of program. The processor configuration might even be changed at runtime, allowing the entire applications-specific system to be reconfigured in response to changing requirements or sensor events.

5. Integration. The program development tools can generate their programs to use a common interface, allowing separately generated programs to be integrated into a complete processing system.

The fundamental reason that application-specific program development tools can do all of this is that work on these tools can be factored over all programs developed for their applications area, instead of having to be done again for each hand-written program. We have developed such tools for two important applications areas for iWarp, namely signal and image processing.

4.1.1. ASSIGN, a program development tools for signal processing

As discussed in Section 3, DSP includes a large class of programs that can be represented as flow graphs: sensor data flows into one or more nodes, and then a series of operations are done, with the processed data flowing from node to node. We are developing a tool, called ASSIGN, for building large DSP applications on iWarp.

ASSIGN inputs a flow graph, which describes a DSP application, and a physical array graph, which describes the topology of the target iWarp array. Each vertex in a flow graph corresponds to a coarse-grained sequential task; each edge corresponds to a buffer between two tasks. Tasks execute an infinite sequence of input, compute, and output actions.

Flow graphs are built from C programs augmented with routines supplied by Assign. Physical array graphs are supplied with the system, although they can be easily constructed with a text editor. ASSIGN maps the flow graph onto the physical array graph, with the goal of maximizing throughput, and then emits a C program for each iWarp cell.

The iWarp pathway and spooling mechanisms allow for a clean and straightforward implementation of flow graphs on the iWarp array. For a given mapping, an edge in the flow graph whose incident vertices are on different iWarp cells is called an ‘‘exposed’’ edge. Each exposed edge in the flow graph corresponds directly to a pathway in the iWarp. Further, the flow control for the FIFO buffer corresponding to each exposed edge is provided for automatically by the iWarp spooling mechanism. Thus, each exposed edge is identified with a pathway connecting the sending and receiving cells.

4.1.2. APPLY and ADAPT, program development tools for image processing

Image processing operations take an image as input, and generate an image or other data structure as output. If the output is also an image, the operation can either be *local* – each output pixel depends on a small corresponding area of the input, or *global* – each output pixel depends on all or a large area of the input. If the output data structure is not an image, the operation is always global.

Local image processing operations can be parallelized by dividing the input and output among processors. This has been done in the APPLY parallel program generator [7]. The APPLY programmer describes the operation to be done more or less as it would be described in the inner loop of a loop over the pixels of the image, except that conceptually the operation happens in parallel on all image pixels at the same time. The APPLY program generator produces an efficient program on iWarp (and a variety of other parallel computers [10]) that executes the operation on an input image. The number of iWarp processing cells and the image size can be varied arbitrarily without modification of the APPLY program.

Global image processing operations can be parallelized using a programming model known at *split and merge*. In split and merge the input image is divided among processors, each of which processes its portion separately, and then the independent results are combined. It can be shown that this model makes it possible to compute any image processing operation that can be computed in forward or reverse order over the image. This model is implemented in the ADAPT program, which produces code for iWarp (as well as other computer systems).

4.2. Single-cell compilers

The parallel program generators are the front-end of the iWarp tool chain. They produce C programs for each cell in the system, which are then compiled by an optimizing single-cell C compiler.

Each cell can also be programmed directly in C or FORTRAN, e.g., for applications in a problem domain for which no suitable program generator exists. The C and FORTRAN compilers share a code generator that incorporates those optimization techniques that were used successfully in an earlier compiler for iWarp [4].

4.3. Execution environment

The core of the execution environment is the Array Cell Executive (ACE). ACE includes a resident runtime system on each cell. The addition of a resident runtime system is a major innovation for systolic computing. As a result, a number of services commonly associated with general purpose microprocessors are now available on iWarp as well. These services include for example a remote debugger and remote I/O. The remote debugger allows the local memory of a cell in the array to be inspected by a debugger executing on a host workstation. Remote I/O allows a file located in either a workstation file system or a file system connected to a peripheral cell to be accessed by an application program executing on a cell in the array.

Two communications services are provided to address both the performance requirements of application programs as well as to create a usable environment for program development.

- Program communication service (PCS) – the application program manages communication explicitly and can therefore perform systolic communication.
- ACE communication service (ACS) – communication is managed by the runtime system. Services include native mode implementation of various message passing protocols, e.g., reliable datagram or request-response.

For example, with PCS, the application program determines the set up and sharing of pathways between programs, and if resources are shared by multiple cells, then the application program has to ensure that no cell causes starvation [6]. By contrast, ACS guarantees that no cell causes starvation by making sure that no cell sends messages that exceed a specific size limit. However, the comfort of the ACS services comes at a price; PCS services incur less overhead (per word sent or received).

5. Summary and concluding remarks

In summary, a number of hardware and software facilities make iWarp a suitable building block for applications-specific computing systems:

- The powerful cells organized as a MIMD architecture allow the implementation of heterogeneous programs, which are often encountered in applications areas such as signal and image processing.
- Hardware supported flow control eliminates the need for expensive programmed synchronization between connecting cells.
- High bandwidth intercell communication supports the very high data rates encountered in signal and image processing.
- Multiple physical I/O busses allows the hardware implementation of a variety of interprocessor topologies, including topologies requiring high numbers of physical links per node.
- Logical channels increase connectivity and deliver guaranteed communication bandwidth.
- The software tool chain eliminates assembly language programming and introduces simplicity, clarity, efficiency, flexibility, portability and integration through parallel program generators, and at the same time increases migration options.

In conclusion, we see that iWarp is the first of a series of building blocks that will revolutionize applications-specific computing systems in the same way that the general microprocessor revolutionized the design of general-purpose computers. The single iWarp chip, integrating computation and communication, eliminates the need for glue logic. The iWarp software, including optimizing compiler and a runtime system, will be shared by all users. As a result, iWarp provides a cost effective building block approach to constructing applications-specific computing systems without sacrificing performance.

Acknowledgments

We appreciate the contributions of the Intel iWarp Microprocessor component team: J. Bauer, B. Bloechel, S. Borkar, B. Breckenridge, M. Camargo, S. Gleason, M. Hester, J. Huang, D. Jenkins, R. Jolly, K. Konigsfeld, G. Meece, B. Moore, D. Nedwek, L. Newby, L. Peake, L. Rankin, K. Self, J. Susman, J. Sutton, K. Troung, J. Urbanski, and the Intel iWarp System Hardware team: R. Carpenter, M. Hill, D. Hofsheier, K. Kerley, D. Manseau, B. McSpadden, and the Intel iWarp Software team: B. Greer, M. Levine, R Maliszewski, H. Mayer, P. Miller, W. Moore, H. Nag, J. Reinders, D. Riss, R. Touzeau, P. Weiss. Special thanks to D. Finan, J. Inouye, H. Lu, H. Pham, and others who helped on the project and to Les Furnanz for his encouragement and review.

References

1. Annaratone, M., Arnould, E., Gross, T., Kung, H. T., Lam, M. S., Menzilcioglu, O., and Webb, J. A. "The Warp Machine: Architecture, Implementation and Performance". *IEEE Trans. on Computers C-36*, 12 (Dec. 1987), 1523-1538.

2. Borkar, S., Cohn, R., Cox, G., Gleason, S., Gross, T., Kung, H. T., Lam, M., Moore, B., Peterson, C., Pieper, J., Rankin, L., Tseng, P. S., Sutton, J., Urbanski, J. and Webb, J. iWarp: An Integrated Solution to High-Speed Parallel Computing. Proceedings Supercomputing ’88, IEEE Computer Society and ACM SIGARCH, Orlando, Florida, November, 1988, pp. 330-339.

3. Borkar, S., Cohn, R., Cox, G., Gross, T., Kung, H. T., Lam, M., Levine, M., Moore, B., Moore, W., Peterson, C., Susman, J., Sutton, J., Urbanski, J. and Webb, J. Supporting Systolic and Memory Communication in iWarp. Proc. 17th Annual Intl. Symposium on Computer Architecture, Seattle, May, 1990, pp. 70-81.

4. Cohn, R., Gross, T., Lam, M., and Tseng, P. S. Architecture and Compiler Tradeoffs for a Wide Instruction Word Microprocessor. Proc. Third. Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS III), Boston, April, 1989, pp. 2 - 14.

5. Gentleman, W.M. and Kung, H.T. Matrix Triangularization by Systolic Arrays. Proceedings of SPIE Symposium, Vol. 298, Real-Time Signal Processing IV, Society of Photo-Optical Instrumentation Engineers, August, 1981, pp. 19-26.

6. Gross, T. Communication in iWarp Systems. Proceedings Supercomputing ’89, November, 1989, pp. 436 - 445.

7. Hamey, L. G. C., Webb, J. A., and Wu, I. C. Low-level Vision on Warp and the Apply Programming Model. In *Parallel Computation and Computers for Artificial Intelligence*, Kluwer Academic Publishers, 1987, pp. 185-199. Edited by J. Kowalik..

8. Menzilcioglu, O., Kung, H. T. and Song, S. W. Comprehensive Evaluation of a Two-Dimensional Configurable Array. Proceedings of the Nineteenth International Symposium on Fault-Tolerant Computing, 1989, pp. 93-100.

9. Printz, H., Kung, H. T. and Mummert, T. Automatic Mapping of Large Signal Processing Systems to a Parallel Machine. Proceedings of the SPIE, Real-Time Signal Processing XII, Society of Photo-Optical Instrumentation Engineers, August, 1989, pp. 2-16.

10. Wallace, R., Webb, J. A., and Wu, I. C. Machine-Independent Image Processing:Performance of Apply on Diverse Architectures. 3rd International. Conf on Supercomputing, May 15-20, 1988, pp. 265 - 276.

Table of Contents

List of Figures