LA-UR 05-1865

Approved for public release;
distribution is unlimited.

Title:	Trident: An FPGA Compiler Framework for Floating-Point Algorithms
Author(s):	Justin L. Tripp Kristopher D. Peterson Jeffrey D. Poznanovic Christine M. Ahrens Maya Gokhale
Submitted to:	International Conference on Field Programmable Logic and Applications (FPL)

Form 836 (8/00)

TRIDENT: AN FPGA COMPILER FRAMEWORK FOR FLOATING-POINT ALGORITHMS

Justin L. Tripp, Kristopher D. Peterson, Christine Ahrens, Jeffrey D. Poznanovic, and Maya B. Gokhale

Los Alamos National Laboratory
Los Alamos, NM 87545
{jtripp, kpeter, cahrens, poznanov, maya}@lanl.gov

ABSTRACT

Trident is a compiler for floating point algorithms written in C, producing circuits in reconfigurable logic that exploit the parallelism available in the input description. Trident automatically extracts parallelism and pipelines loop bodies using conventional compiler optimizations and scheduling techniques. Trident also provides an open framework for experimentation, analysis, and optimization of floating point algorithms on FPGAs and the flexibility to easily integrate custom floating point libraries.

1. INTRODUCTION

Over the past twenty years, Field Programmable Gate Arrays (FPGAs) have been successfully applied to data- and compute-intensive tasks on small fixed point integers, operations prevalent in signal and image processing, cryptography, network packet processing, and bioinformatics. Floating point operations, which dominate supercomputing, have been regarded as too expensive to implement in an FPGA. However, with the Moore's Law growth in FPGA resources, it has become feasible to build high performance floating-point operators on FPGAs [1].

Several floating point libraries [2, 3, 4], applications [5, 6] and application kernels [7, 8] have already been realized in FPGAs. The applications and kernels achieve high performance exceeding that of microprocessors, however, at the cost of hand-coding a custom design in a hardware description language (HDL). Use of a high-level language (HLL) compiler for floating-point could reduce this burden.

Previous HLL work with floating-point and FPGAs consists of analyzing floating point operations for conversion into fixed point operations [9]. This takes advantage of the fixed point operations that can easily be mapped to FPGAs. However, when it is not possible or desirable to convert the operations to fixed point, floating point must be used.

Several HLL Compilers already exist for FPGAs [10, 11]. Celoxica DK [12] and the SRC Map compiler [13] support floating point operations in a limited fashion as external libraries. Most C-to-FPGA compilers do not support floating point data-types and operations. Without rapid prototyping through automatic compilation, exploration of different algorithms and approaches is difficult. An HLL compiler for FPGAs enables designers to experiment with alternative partitioning schemes and quickly determine how an algorithm will perform on a particular FPGA.

We have developed the Trident compiler to provide an open framework for rapid prototyping of floating point algorithms in a HLL. Trident accepts C language input with float and double data types and translates the description into FPGA hardware. Trident allows the user to select from several different floating point libraries [3, 4] or to include a user developed custom floating point library.

2. TRIDENT COMPILER

The Trident compiler builds on and shares code from the SeaCucumber(SC) compiler[14]. Trident extends SC in several new directions: by accepting floating point operations, parsing C input, performing extensive operation scheduling and generating VHDL. Trident also provides a framework for additional compiler optimization and research at different levels of abstraction.

The Trident compiler consists of four principal steps shown in Figure 1. The first step is the LLVM C/C++ front-end. The LLVM (Low Level Virtual Machine) compiler framework[15] is used by Trident to parse input languages and produce a low-level platform independent object code. The second step transforms this low-level object code into the Trident predicated intermediate representation (IR). The Trident IR is further optimized and mapped into a specific hardware library of operations. The third step schedules and pipelines the hardware operations according to the resources available. Finally, the scheduled operations are synthesized into VHDL.

3. LLVM FRONT-END

The Trident Compiler uses the LLVM compiler framework front-end with a few extensions. LLVM's front-end accepts programs written in C, C++ or potentially other languages,

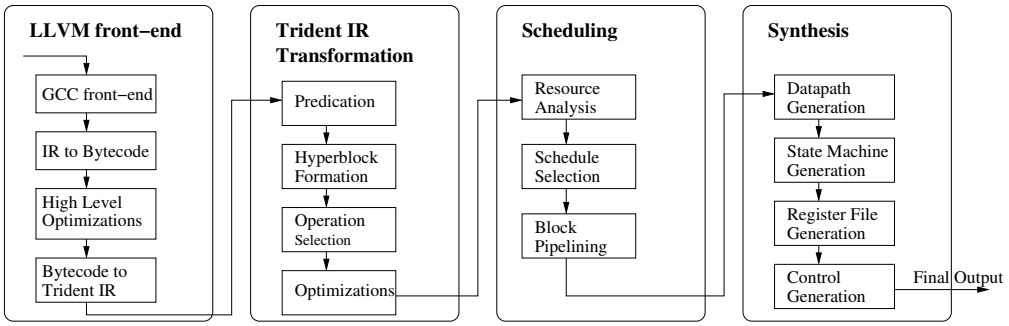


Fig. 1. Four principal steps in the Trident Compiler.

as input and can generate optimized LLVM assembly language. Trident parses the assembly into its own intermediate representation. In the future, we plan to use LLVM to do run-time analysis, since it can provide profiling information.

3.1. Description of LLVM

LLVM was developed at the University of Illinois at Urbana-Champaign to support transparent, lifelong program analysis and transformation for arbitrary programs. LLVM is language-independent and does not require programs or build scripts to be modified, so is transparent to the developer[1]. At the same time, it supports optimization at compile-time, link-time, run-time and offline (idle-time reoptimizer).

The virtual instruction set used by LLVM, the LLVM "bytecode", is a low-level object code representation that uses simple RISC-like instructions, but provides rich, language-independent, type information and dataflow (SSA) information about operands. The information it provides enables LLVM to do sophisticated optimizations, yet its simplicity allows it to be light weight enough to be attached to the executable, thus enabling transformations throughout the program's lifetime.

LLVM's infrastructure supports its strategy for transparency and lifelong program analysis. The LLVM system architecture contains static compiler front-ends that emit code in the LLVM representation. The front-ends can perform language-specific optimizations as well as LLVM passes for global or interprocedural optimizations at the module level. The LLVM bytecode is then combined together by the LLVM linker. At link time, aggressive interprocedural optimizations can be performed. Finally, the resulting LLVM bytecode is translated into native code for a given target. The native code generator inserts profiling instrumentation in order to do run-time optimization. In addition, end-user profile data can be used by an offline optimizer during idle-time.

[1]LLVM is not a high-level virtual machine like Java, for example, with garbage collection. However, optional LLVM components can be used to build high-level virtual machines and other systems that need these services.

3.2. How Trident uses LLVM

LLVM was chosen for Trident because it provides a GCC-based C and C++ front-end which produces machine-independent bytecode. C and C++ are more suitable than some other languages for the supercomputing applications Trident will compile. Machine-independent bytecode is an attractive intermediate representation when the end goal is to synthesize circuits on FPGA hardware.

The first step in compiling a C program in Trident is to use the LLVM gcc to produce the LLVM bytecode. Trident disables optimizations at this time, however does them in a later step. Trident also does not do any linking at this point. It should be noted that the C programs Trident expects, due to its target hardware, should not contain print statements, recursion, **malloc** or **free** calls, function arguments or returned values, calls to functions with variable length argument lists, or arrays without a declared size.

An LLVM Trident pass optimizes the LLVM bytecode using optimization passes provided by LLVM. These passes include constant propagation, small function inlining, loop invariant hoisting, tail call elimination, small loop unrolling, common subexpression elimination, and others. Calling the optimizations from the LLVM Trident pass gives Trident the flexibility of adding or removing some of these optimizations as needed. For example, a loop unrolling pass for operations containing floating point operands has been added.

4. TRIDENT IR TRANSFORMATIONS

The IR transformation step of the Trident Compiler, accomplishes several important tasks. First, operations are converted into predicated form to allow the creation of hyperblocks. Next, further optimizations are performed to remove any unnecessary operations. Finally, all operations are mapped into a specific hardware library selected by the user.

The compile phase parses the output of the LLVM Trident pass into the Trident IR. Trident IR includes predication for every operation and static single assignment (SSA) representations for all operands. The low-level object code representation produced by LLVM provides SSA operands. Trident IR extends this by adding predication for every operation and allowing basic blocks to be expanded into hyperblocks.

Hyperblocks are created by using if-conversion [16] and predication to allow branches to be converted into data-flow operations. A hyperblock, described in [16], is an extended basic-block of instructions which has one input path but may have any number of output paths. The creation of hyperblocks causes the predicated operations to be merged together, leaving only loop-control edges in the control-flow graph. Hyperblocks allow more operations to be considered for concurrent scheduling. This combined with pipelining extracts the available parallelism in the algorithm description.

In addition to the optimizations performed in LLVM, Trident optimizes the code to remove any unnecessary instructions that may be created during if-conversion and hyperblock formation. The optimizations include: common subexpression elimination, dead-code elimination, strength reduction, constant propagation, and alias analysis. These optimizations reduce the number of operations that will be synthesized.

The compiler accomplishes operation selection by mapping a generic set of operations into a particular library. Two different floating point libraries [3, 4] are supported by the Trident compiler. The libraries have different goals, and choosing one allows the user to trade off area, resources, clock speed and latency. Mapping several libraries to a common set of floating point operations gives Trident unique flexibility.

5. RESOURCE ALLOCATION AND SCHEDULING

Floating point data types, with 32- or 64-bit width, present challenges in terms of resource allocation, both on- and off-chip. Floating point modules require significantly more logic blocks on the FPGA than small integer operations. For example, a 64-bit floating point multiply takes 1,274 slices on a Xilinx Virtex 2 (not including the hard multiplier) whereas a representative 32-bit integer multiply occupies 239 slices. The large operand size also implies that more memory bandwidth is required for floating point arrays. Scheduling is further complicated by the interaction with resource allocation.

The resource assignment and array allocation are calculated with respect to the control flow graph (CFG). First, the hardware requirements for the circuit vis-à-vis FPGA resources must be analyzed. Resource sharing of floating point modules as well as assignment of arrays to memory affects scheduling. Second, the operations will be scheduled with one of four different scheduling algorithms: ASAP, ALAP, Force Directed, and for pipelined loops, Iterative Modulo scheduling. The resource allocation problem and its interaction with scheduling are described in more detail below.

5.1. Resource Allocation

Trident addresses two resource allocation problems: allocating space for the logic on the FPGA chip, and allocating arrays to memory. Addressing the first problem, the resource allocation algorithm determines an ideal number of modules, assuming infinite hardware. If the user chooses to minimize area, Trident creates enough modules for the maximum used in a single cycle (which is found by performing a preliminary schedule, assuming infinite area). For example, if the circuit has five additions, but has at most three in any cycle, only three adders will be created. If the ideal number of modules does not fit on the FPGA, Trident uses a heuristic to find the minimum number of modules to share, which in turn minimizes execution slowdown.

The heuristic resource allocation algorithm (Fig. 2(a)) first re-uses those modules with the greatest latency and, in case of equal latencies, the largest module. To minimize resource sharing, the algorithm first checks to see if sharing only one of the chosen modules will sufficiently reduce space requirements, and if not it will reduce the number of this module type available by half. This process is repeated until there is only one module left or the space constraints have been met. If there is still not enough room, Trident will share the next slowest or biggest module.

```
foreach(hyperblock)
  foreach(operation in hyperblock)
    sum area requirements for all
       operators and save count
       of each operator used
  if resource requirements too large
    sort operations in hyperblock
      by latency then area
    while requirements are too large
      foreach(operation in hyperblock)
        share one resource if possible
        else reduce by half the # of available
         resources
```

(a) Share Resources

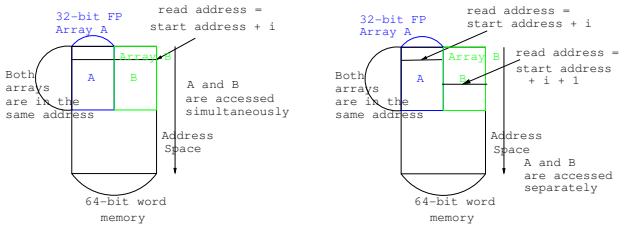
```
search for minimum array allocation cost:
while(not all arrays allocated)
  for each unscheduled array
    for each memory
      if array read/write req != mem
                        read/write permissions
        next mem
      if array size > mem size
        next mem
      if space found
        calculate cost
      if number of tries to allocate this array
                                  – cost == 0
        allocate array to this memory
    if array not allocated
      increment try count
```

(b) Memory Allocation

Fig. 2. Pseudo-code for resource sharing and memory allocation.

The second problem with floating point computation is the bandwidth requirements due to large operand size when communicating with memory. For example, if there are four 64-bit word external memories available, each with one read bus, we can read at most eight 32-bit floating point val-

ues during a single clock tick. Since data arrays are often too large for on-chip memory, external memory bandwidth is the primary factor limiting parallelism within the circuit. For example, if the target FPGA board has four 64-bit word external memories with one bus each, it can access only 8 32-bit floating point data words per clock cycle. If there is sufficient parallelism in the computation to use more than 8 words per clock cycle, it cannot be exploited. In addition, when memory accesses are ordered sequentially, the circuit may need additional pipeline registers and incur additional latency.



(a) Packed Arrays (b) Unpacked Arrays

Fig. 3. Figure showing packed and unpacked arrays.

Besides bandwidth limitation to memory, the pattern of array allocation to memory influences throughput [17, 18]. Three factors can limit memory throughput: First, the maximum number of reads or writes to a given memory during a cycle. To help determine this, prior to array allocation, Trident performs a preliminary schedule of the design assuming infinite memory communication bandwidth. Knowing this, the maximum number of data accesses can be calculated for any given pattern of array to memory allocations. The second major factor is the number of read and write buses to external memories. The third major factor affecting throughput is the possibility of packing arrays into the same address space. For example, if a memory has 64-bit data words, and the arrays use 32-bit words, it is possible to fit two arrays in the same memory location and access both arrays simultaneously with one read or write operation. This is only possible if the arrays are accessed with the same index and the same read or write operation is performed (Fig. 3).

The impact of these three factors will be referred to as the "cost" of a given array to memory allocation pattern, where the cost equals the number of extra cycles necessary to perform memory access than what was used in the preliminary schedule. The heuristic used by Trident (Fig. 2(b)) to search for the minimum cost, iterates through the list of arrays, matching them individually with each memory. If the read/write permissions for this memory and its available space allow placing the array in this memory, the heuristic attempts to find a place for the array in memory, attempting to pack it with other arrays where possible. If a place is found, the cost for this allocation using the three factors described above, is calculated. If the cost is zero, the array is placed in this memory and otherwise the algorithm continues to the next memory. If no place is found, the algorithm proceeds to the next array, but counts the number of attempts to allocate the previous array. During future attempts to allocate an array, the cost is subtracted from the number of tries. When this difference reaches zero, the array will be allocated. This way, all arrays that can be allocated with a cost of zero will be allocated first, followed by all those with a cost of one, and so on. The heuristic loops until all arrays have been allocated or memory is full.

5.2. Scheduler

The trident compiler provides four different schedulers for scheduling operations. Non-loop blocks are scheduled using ASAP, ALAP, and Force-Directed [19]. Iterative Modulo [20] scheduling is applied to loop blocks to schedule the block and calculate the Initiation Interval (II) for pipelining the loop.

If hardware resources are limited, the scheduler must take this into account and determine which operations should be shared. If modules must be shared the operations sharing a module cannot start in the same cycle. It is desirable to assign operations to share a module if they can be scheduled at different times. If this is not possible, the start of these operations must be staggered over multiple cycles. This limitation becomes another criteria along with data dependencies that sets the soonest limit for ASAP, the latest limit for ALAP, and an illegal slot for force-directed and modulo scheduling. Multiple accesses to a memory must also be spread out as constrained by data dependencies and memory communication bandwidth (i.e., the number of buses).

Although Trident uses predicated operations, the scheduler considers predicates only when the result is stored to a register or memory. This implies that operations may be executed speculatively. However, since the operation must be built in the FPGA, this cost is mitigated.

Scheduling for a circuit introduces issues not applicable in a processor. For example, a processor has a fixed set of registers for use by all operations. On the other hand, the registers used in a circuit must be created. Also, data in a circuit can be transmitted by wire between modules, but only if the signal will not last multiple cycles or be transfered between multiple hyperblocks. Different iterations of a loop behave like separate hyperblocks. This becomes an issue especially when implementing modulo scheduling.

In modulo scheduling, the resultant block of operations, contains the equivalent of multiple iterations of the loop with staggered starts. This modulo block is a sliding window of operations of the fully unrolled loop. Each complete execution of an iteration of the modulo block executes the same

number of operations as the original loop, but speedup occurs since multiple iterations of the original loop are running simultaneously although in different stages.

At the end of each fraction of the original loop, the resultant data must be saved in a register and reread when the modulo block starts a new iteration. This necessitates the insertion of extra register read and write operations into the body of the original loop. To do this, Trident checks if two successive operations occur in different iterations of the modulo block. If so, the parent's results must be stored to and loaded from a register and wired to the child with a new wire to ensure the child is always using data from its same iteration of the large loop. Since, operations can be scheduled and unscheduled multiple times in modulo scheduling, sometimes parents and their children are in the same iteration of the modulo block and sometimes not. Therefore instead of always placing a load and store between operations, Trident ensures there is always time for a load between cycle zero and the beginning of the operation and between its finish and cycle II-1, without checking if a load or store is necessary due to a parent or child being in a different iteration. The loads and stores are inserted after scheduling is complete.

6. SYNTHESIS

After the scheduler finishes, the control flow graph has been modified to include timing information which is then passed to the synthesizer. Three major stages occur in the Trident synthesizer: library mapping, abstract design generation, and back-end code generation.

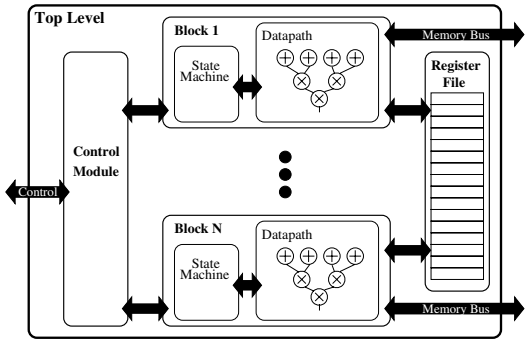


Fig. 4. Abstract circuit design hierarchy

The abstract circuit structure, shown in Figure 4, contributes to Trident's flexibility. It enables the compiler to generate a top-level blueprint for the design while leaving the underlying technology open until the final code generation stage. The underlying technology consists of the target hardware description language, the target floating point cores, and the target hardware architecture.

Hierarchy is utilized in the abstract circuit structure in order to preserve the modularity. The top-level of the abstract circuit contains subcircuits for each block in the control flow graph input and also a single register file. The register file is shared by all block subcircuits. Each block subcircuit contains a state machine and a datapath subcircuit. The state machine is determined by the initiation interval of the design, and it controls the timing of the block's datapath.

The datapath implements the logic needed to represent the flow of data through all of the operations found in the control flow graph. It contains operators, predicate logic, local registers and wires that connect all of the components. If the target is a pipelined design, pipeline registers are added between operators in the datapath in order to preserve the correct data flow behavior.

Trident's internal technology-independent circuit representation allows for targeting multiple hardware description languages at the back-end generation stage. Currently, Trident targets VHDL, the common hardware description language. In addition, Trident can output a file that can be used to visually debug the structure of the design. The simplicity of the back-end generation stage enables relatively simple additions to the list of target technologies.

Each target technology's back-end generator implements the abstract circuit generator. Whenever an abstract component is generated, a request is made to generate it in the target technology. If the request can be granted, the component is created in the chosen technology. This process is continued for each abstract component until the target design is complete.

Trident supports the Cray XD1 as a reconfigurable synthesis target[21]. In order to integrate a Trident-generated design with the XD1 architecture, application-specific logic is created to connect the application's memory uses with the XD1's memory bus interface. This logic, consisting of a series of address decoders and guarding muxes, controls how data flows to and from memory.

7. CONCLUSIONS

Trident provides an open framework for exploration of floating-point libraries and computation in FPGAs. The Trident framework allows for user optimizations to be added at several levels of abstraction. Users can also trade-off clock speed and latency through their selection of different floating-point libraries and optimizations. Performance is achieved through parallelism extraction at both the operation level and the hyperblock level by using predication and pipelining. Operation scheduling is aware of the hardware constraints existing in a particular system and uses several techniques to take advantage of the available bandwidth.

Trident is still a work in progress. Currently, the tool supports simulation and synthesis for two floating point li-

braries: Quixilica, and Arénaire FPLibrary. In addition, several simple applications can be synthesized and executed in the FPGA on the XD1 and we are currently working to increase the subset of C that can be synthesized and enhance performance.

8. REFERENCES

[1] K. Underwood, "FPGAs vs. CPUs: Trends in peak floating-point performance," in *ACM/SIGDA Twelfth ACM Int'l Symposium on Field-Programmable Gate Arrays (FPGA 2004)*, 2004.

[2] P. Belanović and M. Leeser, "A library of parameterized floating-point modules and their use," in *FPL 2002: The 12th International Conference on Field-Programmable Logic and Applications*. Springer-Verlag, 2002, pp. 657–666.

[3] J. Detrey and F. de Dinechin, "FPLibrary, a VHDL library of parametrisable floating-point and LNS operators for FPGA," http://perso.ens-lyon.fr/jeremie.detrey/FPLibrary/, 2004.

[4] QinetiQ Holdings Ltd., "Real time systems lab," http://www.quixilica.com/ products.htm, 2002.

[5] M. Gokhale, J. Frigo, C. Ahrens, J. L. Tripp, and R. Minnich, "Monte carlo radiative heat transfer simulation on a reconfigurable computer," in *International Conference on Field Programmable Logic and Applications*, August 2004.

[6] J. P. Durbano, F. E. Ortiz, J. R. Humphrey, P. F. Curt, and D. W. Prather, "FPGA-based acceleration of 3D finite-difference time-domain method," in *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa, CA, 2004.

[7] *Area and Power Performance Analysis of Floating-point based Application on FPGAs*. Lexington, MA: Seventh Annual Workshop on High Performance Embedded Computing (HPEC 2003), September 2003.

[8] S. Choi and V. Prasanna, "Time and Energy Efficient Matrix Factorization using FPGAs," in *FPL 03: 13th International Conference on Field Programmable Logic and Applications*, Sept. 2003.

[9] P. Banerjee, D. Bagchi, M. Haldar, A. Nayak, V. Kim, and R. Uribe, "Automatic converstion of floating point matlab programs into fixed point FPGA based hardware design," in *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa, CA, 2003, pp. 263–266.

[10] M. B. Gokhale, J. M. Stone, J. Arnold, and M. Kalinowski, "Stream-oriented fpga computing in the streams-c high level language," in *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa, CA, 2000.

[11] S. C. Goldstein, M. Budiu, M. Mishra, and G. Venkataramani, "Reconfigurable computing and electronic nanotechnology," in *Proceedings of the IEEE 14th Int'l Conference on Application-specific Systems, Architectures and Processors*, IEEE. The Hague, The Netherlands: IEEE Computer Society, June 2003.

[12] L. Celoxica, "HANDEL-C language overview," Celoxica, Ltd, Tech. Rep., Aug 2002.

[13] M. C. Smith, J. S. Vetter, and X. Liang, "Accelerating scientific applications with the src-6 reconfigurable computer: Methodologies and analysis," 2005, presentation on the SRC-6 Reconfigurable Computer.

[14] J. L. Tripp, P. A. Jackson, and B. L. Hutchings, "Sea cucumber: A syntheiszing compiler for fpgas," in *Field Programmable Logic and Applications: 12th Int'l Conference Proceedings / FPL 2002*. Montpellier, France: Springer-Verlag, 2002, p. n/a.

[15] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the 2004 Int'l Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, CA, Mar 2004.

[16] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann, "Effective compiler support for predicated execution using the hyperblock," in *25th Annual Int'l Symposium on Microarchitecture*, 1992.

[17] M. B. Gokhale and J. M. Stone, "Automatic allocation of arrays to memories," in *IEEE Symposium on Field-Programmable Custom Computing Machines*. Napa, CA: IEEE Computer Society Press, 1999, pp. 63–69.

[18] H. Lange and A. Koch, "Memory access schemes for configurable processors," in *Field Programmable Logic and Applications: 12th Int'l Conference Proceedings / FPL 2000*. Springer-Verlag, 2000.

[19] P. Paulin and J. Knight, "Force-directed scheduling in automatic data path synthesis," in *24th Design Automation Conference*, June 1987, pp. 195–202.

[20] B. R. Rau., "Iterative modulo scheduling." *Int'l Journal of Parallel Processing*, vol. 24, pp. 3–64, 1996.

[21] *Cray XD1 Datasheet*, Cray Inc., Seattle, WA, September 2004.