Designing a Coprocessor for Recurrent Computations *

Kumar N. Ganapathy and Benjamin W. Wah
Coordinated Science Laboratory
1308, West Main Street
Urbana, IL 61801.
e-mail: {kumar,wah}@manip.crhc.uiuc.edu

Abstract

In this paper, we present the design of an application-specific coprocessor for algorithms that can be modeled as uniform recurrences or "uniformized" affine recurrences. The coprocessor has a regular array of processors connected to an access-unit for intermediate storage of data. The distinguishing feature of our approach is that we assume the coprocessor to be interfaced to a standard, slow (single-ported) memory with low bandwidth. Hence, good performance is achieved by effectively exploiting data locality in the applications by the compiler, and the final architecture is chosen by a tradeoff analysis driven by the mapping process. Preliminary results indicate that the coprocessor has significantly lower clock rates or higher performance than that of existing RISC processors and is cost-effective for executing loop computations.

1 Introduction

In the last decade, application-specific architectures have been recognized as a promising approach to high-performance computing. These special-purpose processors have high-performance due to a combination of high degree of pipelining, parallelism and reduced memory contention by multiple use of each data item. The *goal* of this research is to develop a cost-effective, high performance VLSI coprocessor that is programmable for an application or an application domain. The coprocessor must interface with a front end that is responsible for data input and control. It should be characterized by low system cost, a small size and performance that is significantly better for target applications than a general purpose computer of similar cost. Our application-specific coprocessor can be viewed as a back-end system to accelerate loop computations in high-end workstations, or as a VLSI pipeline in supercomputers similar to a vector functional unit for accelerating vector instructions.

There have been numerous efforts to develop general purpose systolic computers in the past few years. These include iWarp [1], Matrix-1, SLAPP, VATA [2], medium-grain architecture for image and signal processing [3], and a host of others. However, most of these designs have powerful processing elements with large local memories, and high-bandwidth data interconnect between the processors and the host/global memory. Hence, the cost of such systems is very high (in thousands/millions of dollars) due to expensive hardware and complex designs. In our approach, we aim at building a simple, resource-limited VLSI array processor that can be attached to standard single-ported main memories (with long latencies for data access), resulting in a system with low cost and acceptable performance for targeted applications. We believe that good performance can be obtained by sophisticated mapping and analysis by the compiler that exploits the hardware effectively for the given set of applications. Hence, there is virtually no programming effort for the user, as the compiler accepts a high-level description of the algorithm and generates address sequences and control programs to carry out the required computations. This is in contrast to most existing "class-specific" or general-purpose systolic architectures where it is the programmer's responsibility to determine which computations are performed by which cells, and how data is moved among the cells and memory.

Recently, there have been efforts on developing a systematic approach to partition and map matrix algorithms on mesh-connected arrays [4]. However, our approach differs in trying to extract maximum data re-use under the constraint of a fixed low-bandwidth interconnect to main memory in our array proces-

*Research Supported by Joint Services Electronics Program contract N00014-90-J-1270, National Science Foundation grant MIP 92-18715, and an IBM graduate fellowship grant.
Proc. 5th IEEE Symposium on Parallel and Distributed Processing, December 1993.

sor. This work also differs from traditional work on systolic-array mapping/partitioning on to fixed processor arrays [5, 2] in which only a limited storage in the processor array is assumed, and only the effect of main-memory latency due to low-bandwidth interconnection to main memory is considered. The goals of our design that are different from other research efforts are: (i) combination of hardware and compiler methods (Section 4), (ii) reduction in clock rate for the same throughput (Section 5), and (iii) fixed-bandwidth interconnect to main memory (Section 2)

The remainder of this paper is organized as follows. We first present our proposed array processor (Section 2), describe the target algorithms (Section 3) and the proposed mapping technique (Section 4), and finally evaluate and discuss our results (Section 5) using matrix product as an example.

2 Coprocessor Architecture

The architecture proposed in this research is broadly composed of the following components (Figure 1).

Memory (MM). The memory is a standard memory (usually interleaved) with one memory-access port for storing all the data (inputs and outputs) involved in the computations. The time to access the data is relatively long because of the decoding required as well as the limited bandwidth (Von-Neumann bottleneck).

Access-Unit (AU). The purpose of the access-unit is to supply data to the processor array (PA) and mask the long main-memory latencies. The fixed amount of storage in the AU is organized as FIFO queues (or LIFO stacks), and explicit memory addresses (except for queue or stack numbers) are not used. In each cycle, the data present at the head of the queues are sent into the PA or the memory through the output network (Figure 1), and data from the PA or the memory are sent into the top of the queues. This implicit addressing saves on address computations and also allows the AU to be built with a larger number of I/O ports (as compared to a random access memory like a cache with explicit addressing).

The AU can also perform more complex operations such as (i) prefetch data from the main memory into its queues to hide the memory latency, (ii) reorder the data before sending them to the PA, and move the data through the queues at different clock rates for the queues, (iii) perform indirect addressing of main memory, where a sequence of addresses obtained from the memory are subsequently used to access the data in the main memory. The indirect addressing corresponds to gather-scatter instructions used to process sparse matrices in a vector supercomputer.

Processor Array (PA). The processor array obtains data items from the head of the queues in the AU. The PA is I/O bounded and has regular pipelines of processing elements (PE). For instance, the PEs can be organized as a linear array, a set of linear arrays, or a 2-D mesh. Each of the PEs is locally connected and can only perform near-neighbor communication. The array organization with a small number of boundary (I/O) processors reduces the demand on the AU, and simplifies the design of the network between the AU and PA.

The design of the coprocessor includes *both* the hardware and the compiler, as good performance is dependent on the ability of the compiler to exploit existing hardware, as well as the choice of the hardware that can be best supported by the compiler. Our design procedure has two major steps. (i) We first define a model of the architecture in terms of a few parameters (abstract model). (ii) We then arrive at the final architecture by a trade-off analysis driven by the compiler. The architecture is defined by the following parameters: (i) number of PEs (#PE), (ii) local memory in each PE (l), (iii) bandwidth between the access-unit and the main memory (B_{MM}), (iv) size of the access unit (p), and (v) interconnection topology of the PEs (linear or square). The final architecture is obtained after an analysis of performance versus cost from the point of view of software, algorithms, technology and performance constraints.

3 Algorithm Model

A large number of applications in image processing, digital signal processing and other scientific applications that require regular compute-intensive operations can be modeled as affine dependence algorithms. Nested loops are often the most time-consuming kernels of these programs and are the target of our coprocessor based on processor arrays. A large number of these nested-loop programs can be modeled as affine recurrences. For instance, the following Fortran-like nested loops can be represented by a system of recurrence equations.

$$\begin{array}{l} \text{DO } (j_1 = l_1, u_1 \; ; \; j_2 = l_2, u_2 \; ; \; \cdots \; ; \; j_n = l_n, u_n) \\ \quad S_1(\overline{J}) \; ; \\ \quad S_2(\overline{J}) \; ; \\ \quad \vdots \end{array}$$

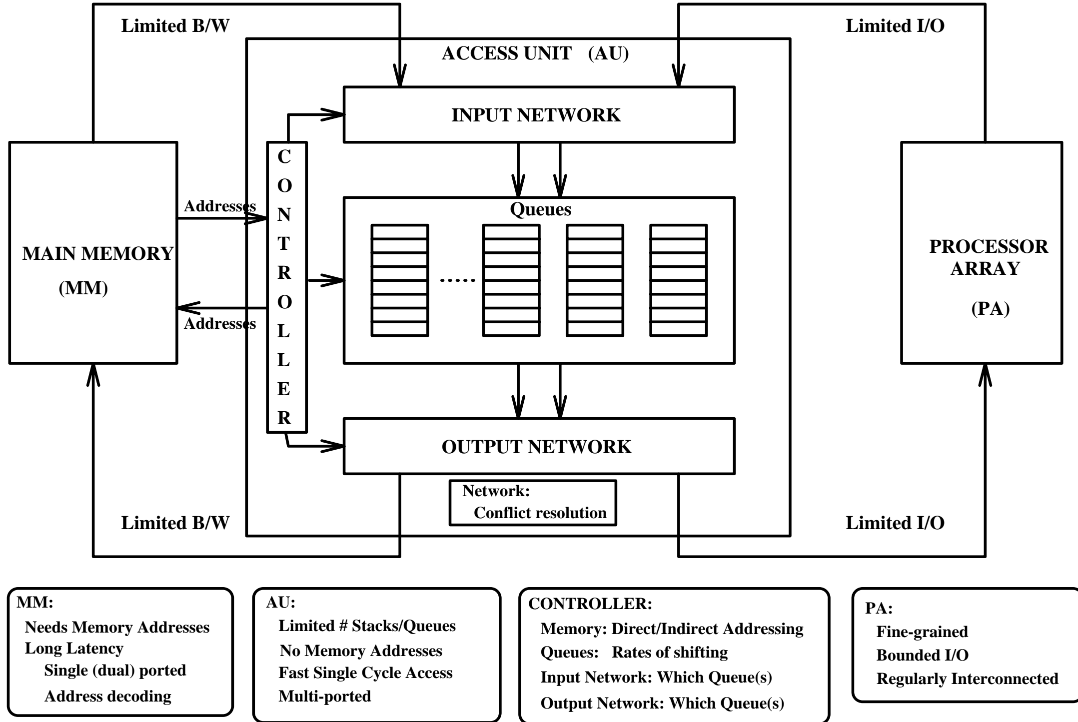

Figure 1: Proposed architecture for solving a class of algorithms modeled by recurrences

$S_t(\overline{J})$;
END

The column vector $\overline{J} = [j_1, j_2, \cdots, j_n]^T$ is the index vector (or index point). $S_i(\overline{J})$, $i = 1, \cdots, t$ are t assignment statements in iteration $\overline{J}$ having the form

$$Z_i(y(\overline{J})) = \phi\left[Z_1(x_1(\overline{J})), \cdots, Z_r(x_r(\overline{J}))\right], \qquad (1)$$

where $1 \leq i \leq r$. If all loop bounds l_i and u_i are affine functions of loop variables $j_1, \ldots, j_{i-1}$, and indexing functions $y()$ and $x_k()$, $k = 1, \cdots, r$, are affine functions of the form $A\overline{J} + \overline{d}$, then affine recurrence equations (ARE) with a convex polyhedral domain can be used to model the above program.

Uniform dependence algorithms or uniform recurrence equations (URE) form a sub-class of AREs, where the indexing functions $y()$ and $x_k()$ are of the form $\overline{J} - \overline{d}$ (matrix A is the identity matrix now), and $\overline{d}$ is a constant vector of n elements. Uniform dependence algorithms can be found in many scientific computations, digital signal processing and other fields.

There exist techniques [6] to transform AREs to UREs where the affine dependencies are "uniformized." The basic idea is to select a few basic integral vectors (which are the uniform dependencies) such that all affine dependencies of the ARE can be expressed as non-negative integer linear combinations of the basis vectors. This uniformization also removes the undesirable broadcasts of data in a VLSI PA.

In this article, we focus on algorithms that can be modeled as uniform recurrences and affine recurrences that can be uniformized. Hence, the starting point of mapping assumes a convex polyhedral domain and a set of constant dependence vectors collected into a matrix called the dependence matrix D.

Example 1. Matrix Multiplication. Consider the matrix multiplication $Z = X.Y$ of two $N \times N$ matrices X and Y.

DO $(i = 1, N \ ; \ j = 1, N \ ; \ k = 1, N)$
  $z(i, j) = z(i, j) + x(i, k)y(k, j)$
END

Eliminating broadcasts and "pipelining" the variables we get,

DO $(i = 1, N \ ; \ j = 1, N \ ; \ k = 1, N)$
  $X(i, j, k) = X(i, j - 1, k)$
  $Y(i, j, k) = Y(i - 1, j, k)$

$Z(i,j,k) = Z(i,j,k-1) + X(i,j,k)Y(i,j,k)$
END

The dependence vectors collected into a matrix are

$$D = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$
$$\quad\quad y \quad x \quad z$$

4 Mapping UREs to Processor Array Architecture

The goal of the mapping or compilation process is to generate address sequences for the controller in the access-unit and code sequences that the PEs execute on receiving the data. The mapping process is broken down into the following five steps.

1. Detect and extract the dependence graph (DG) from the high-level program.
2. Partition the DG into blocks that can be executed in a single pass of the data on the processor array.
3. Sequence blocks through the access-unit, *i.e.,* which blocks will occupy the access-unit at each time step.
4. Schedule execution of a single block on the processor array.
5. Generate address and code sequences from Steps 3 and 4.

Since the first step is well-studied and exists in most compilers [7], we proceed with the description of remaining steps in the mapping process.

4.1 Partitioning

The objective in this step of the mapping process is to partition the DG into non-overlapping blocks or chunks that can be processed by the PA in one pass. This step is necessary because the PA has only a limited number of PEs and I/O pins. Hence, the DG is broken down into blocks for which the PA is "full-sized" and can process them very efficiently.

In this paper, our approach to partitioning is similar to that of Moldovan and Fortes [5] in which, for an n-D algorithm, we try to find n-independent hyperplanes to partition the DG into blocks. Hence, our blocks are "parallelopipeds," and the shape of the blocks can be described by the partitioning matrix P consisting of n partitioning vectors.

$$P = [p_1\, p_2\, p_3\, \cdots\, p_n]$$

The following lemma given without proof presents the conditions for the validity of a partitioning matrix P. The idea used here is that all the dependence vectors should cross the partitioning hyperplanes in the same direction.

Lemma 1. *The partitioning of the DG by a partitioning matrix P is valid if and only if $P^tD \geq \overline{0}$ or $P^tD \leq \overline{0}$, where D is the dependency matrix.*

Next, we present a procedure for choosing a good partitioning vector that results in very little communication between adjacent dependent blocks.

Let $g = rank(D)$, where $D = [d_1 d_2 \cdots d_k]$ is the dependency matrix. Hence, only g of the k dependence vectors are linearly independent. Without loss of generality, assume that the first g columns are linearly independent, and let $D^{'} = [d_1 d_2 \cdots d_g]$ be an $n \times g$ matrix consisting of the g linearly independent vectors of D. Let $D_i^{'}$ be an $n \times g - 1$ matrix derived from $D^{'}$ by dropping the i-th column vector, *i.e.,* $D_i^{'} = [d_1 \cdots d_{i-1} d_{i+1} \cdots d_g]$. The number of partitioning hyperplanes needed to partition the DG is g (since $g = rank(D)$). Hence, matrix $P = [p_1 \cdots p_g]$ is chosen such that p_i is given by

$$\begin{aligned} p_i^t\, D_i^{'} &= \overline{0},\ 1 \leq i \leq g, \ \text{ and} \\ p_i^t\, d_i &> 0. \end{aligned} \tag{2}$$

The idea is to choose p_i as the basis vector of the left null space of matrix $D_i^{'}$, and invert the sign of the elements of p_i if $p_i^t d_i < 0$. Hence, by construction, the partitioning matrix P is feasible.

Example 2. For 3-D matrix multiplication, the partitioning matrix P is equal to D, as $D^tD = I_3 > \overline{0}$ (I_3 is the 3-D identity matrix).

The choice of partitioning matrix P by the above procedure results in the minimum amount of communication between blocks in the partitioned DG. The total amount of communication c from a (interior) block is given by $c = \sum_{i=1}^{g} \frac{\prod_{i=1}^{g} b_i}{b_i} min(b_i, p_i^t d_i)$, where b_i is the size of the block along p_i; *i.e.,* two partitioning hyperplanes perpendicular to p_i are placed b_i apart. The complexity of finding matrix P is $\begin{pmatrix} k \\ g \end{pmatrix} O(n.g^2)$,

as there are $\binom{k}{g}$ ways of choosing g independent columns of D, and $O(n.g^2)$ is the cost of finding a null-space vector of an $n \times (g-1)$ matrix.

4.2 Sequencing blocks through the AU

The execution model of the coprocessor array is as follows. Initially, p blocks from main memory (MM) are loaded into the AU to be processed by the PA. As the execution proceeds, new blocks are fetched from the MM into the AU, and some of the existing blocks in the AU are written back to MM. Since the AU is of limited size, a block of data will have to be fetched multiple times from the MM, and the goal in this step is to decide which blocks will be fetched into the AU as the execution proceeds. The choice of the blocks that will reside in the AU must be done to reduce the traffic between MM and AU; equivalently, data reuse should be maximized for the blocks in the AU. We can think of the AU forming a "storage window" (or tile) over the DG of the algorithm, and the output of this phase describes how the "storage window" will be moved over time over the DG in a non-overlapped fashion. (Otherwise, some computations will be redundant). In this paper, due to space limitations, we only describe the block sequencing for the matrix-product example. The general scheme can be obtained by a simple extension of the scheme presented for arbitrary dependence vectors.

Example 3. Consider a 3-D mesh with $N_1 = N_2 = N_3 = 6$ and $p = 9$; *i.e.,* the storage window or tile is a square of size 3×3. Consider the compressed DG formed by shrinking each block in the DG to a node. The ordering of the tiles that dictates the block movement is shown in Figure 2. The number of accesses to main memory (or the I/O complexity) of the above scheme is given by $Q = 2\,\frac{N^3}{\sqrt{p}} + N^2$

The above scheme can be proved to be *optimal* with respect to the number of accesses to the main memory. It has been shown that for n-D-mesh DGs, $Q = \Omega\left(\frac{V}{\sqrt[n-1]{S}}\right)$, where S is the size of the limited memory and Q is the I/O complexity [8].

4.3 Executing a block in the PA

The goal in this step is to map a single block on to the PA so that it can be executed in a single pass. For this step, we use the *Generalized Parameter Method* (GPM) [9], an optimal parameter-based method we have developed earlier, to map a given block on to

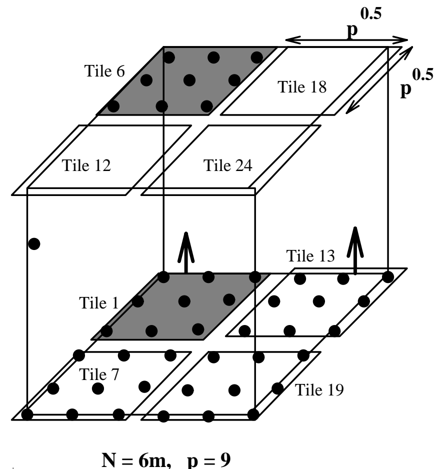

Figure 2: Compressed DG for matrix multiplication. The shaded area shows the storage window (blocks in AU). For $N = 6m$ and $p = 9$, there are 24 tiles, and the ordering of the tiles defines how the data are brought into the AU.

the processor array for execution. GPM outputs the space-time description of the block (which data should be input at each time step) for correct execution. The next paragraph describes this step informally.

Given a block of the DG, GPM finds a feasible space-time mapping of the block and the input-data distributions. Basically, a data distribution is similar to a reservation table that describes the utilization of the processors over different instants of time. The maximum overlap between adjacent blocks is simply computed by moving a copy of the data distribution (or reservation table) over itself until no collisions occur. The two steps of determining a feasible mapping using GPM and computing the maximum overlap using the reservation table are iterated until a suitable space-time mapping with maximum overlap between adjacent blocks is found. If B is the size of the block, the worst-case time for this procedure is $O(B^2)$.

Having presented a glimpse at the individual steps of the mapping process, Figure 3 gives the overview of the steps involved in the compilation process. There are two iterative flows in the mapping process. (i) Edge e_1 represents loops over GPM to derive feasible data distributions for a given block. Each iteration of this loop represents a feasible choice for the PA that can process the individual blocks. (ii) Edge e_2 represents loops over different block sizes and shapes in Step 1.

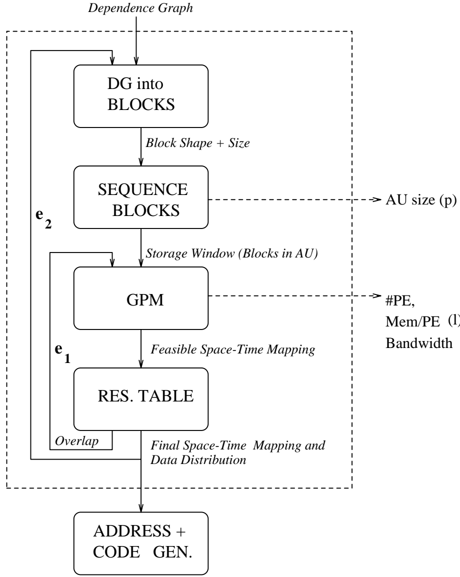

Figure 3: Mapping a DG to the PA

5 Results

In this section, we present our preliminary results in mapping algorithms described by UREs by studying the problem of 3-D matrix multiplication, a benchmark in this class. A lot of digital signal processors available today are evaluated by their performance on FIR Filtering (which are equivalent to matrix-vector multiplications). The DG is an $N \times N \times N$ cubical mesh. The results for the matrix-multiplication example can be extended directly to other uniform recurrences (any DG of an n-D URE can be transformed into an n-D mesh by a linear transformation, and the results for 3-D meshes are applicable to n-D meshes by direct dimension extension). As presented in Section 4.1, the 3-D-mesh DG is partitioned into cubical blocks of size $m \times m \times m$.

5.1 Performance Measures

The fundamental metric we use is the performance of the targeted algorithms on a given amount of silicon chip area. Hence, completion time and area estimate in terms of the abstract model parameters is required.

The total area occupied by the array architecture is the sum of the areas of the processors, AU, controller, and interconnection networks between the AU-MM and AU-PA.

$$\begin{aligned} Area &= Area_{proc} + Area_{AU} + Area_{pins} + \\ & \quad Area_{controller} + Area_{network} \end{aligned} \tag{3}$$

Since the dominant terms are $Area_{proc}$ and $Area_{AU}$, as a first-order approximation, the area index is given by

$$\text{Area Index} = \underbrace{\#PE(Area_{PE} + l)}_{Area_{PA}} + \underbrace{3\sqrt{p}m^2 + pm^2}_{Area_{AU}} \tag{4}$$

where $Area_{PE}$ is the area of a single PE in memory words; *i.e.,* it captures the layout cost of a PE, and l reflects the local memory per PE. Since the AU holds $\sqrt{p} \times \sqrt{p}$ blocks of the DG (Figure 2), storage is needed for (i) pm^2 elements of Z, (ii) $\sqrt{p}m^2$ elements of X, (iii) $\sqrt{p}m^2$ elements of Y, and (iv) additional $\sqrt{p}m^2$ words for the next set of $\sqrt{p} \times \sqrt{p}$ blocks from MM.

The total completion time T_{compl} in PE-cycles for executing matrix product is given by

$$\begin{aligned} T_{compl} &= \underbrace{max\left(\frac{2\sqrt{p}m^2}{B_{MM}}, p\, t_{pipe}\right)\frac{N^3}{pm^3}}_{\text{All blocks}} \\ &+ \underbrace{\frac{2pm^2}{B_{MM}}\left(\frac{N^2}{pm^2} - 1\right)}_{\text{XY plane except block (0,0,0)}} + \underbrace{\frac{pm^2 + 2\sqrt{p}m^2}{B_{MM}}}_{\text{Load Time: block (0,0,0)}} \\ &+ \underbrace{max\left(\frac{4pm^2}{B_{MM}}, p\, t_{pipe}\right)}_{\text{Drain Time: block (N,N,N)}} \end{aligned} \tag{5}$$

where B_{MM} is the bandwidth between MM-AU in elements per PE-clock. The first term in Eq. 5 is the dominant term and is equal to the product of the time taken for each window (with $\sqrt{p} \times \sqrt{p}$ blocks) and the number of windows over the DG. For each window of $\sqrt{p} \times \sqrt{p}$ blocks, $2\sqrt{p}m^2 B_{MM}$ is the time it takes to fetch the elements of X and Y needed for the next window, and t_{pipe} is the time to process a block, which is the sum of the computation time and the non-overlapped load and drain times. The second term in the equation models the additional time required whenever the window returns to the XY plane; this involves writing and reading pm^2 elements of product matrix Z.

In order to have efficient processing, the access unit must be large enough to completely overlap the loading of the inputs of the next window with the processing of the current one. Therefore,

$$\frac{2\sqrt{p}m^2}{B_{MM}} \leq p\, t_{pipe} \Longrightarrow p \geq \left(\frac{2m^2}{B_{MM}t_{pipe}}\right)^2 \tag{6}$$

The above value of p indicates the size of AU required to mask the effect of long memory latency completely.

Example 4. Let $N = 512$, $m = 8$, $B_{MM} = 1/5$ (5 cycles per word) and $\#PE = m^2 = 64$. For a square array, $t_{pipe} = m = 8$, and the AU size should be about $409.6K$ words or $13.1M$ bits assuming 32 bits/word. For a linear array, $t_{pipe} = m^2 = 64$, and the AU size is about $8K$ words and $256K$ bits. Although $13M$ bits of fast memory in the AU is not possible with the current technology, $256K$ bits is very feasible. The reduction in performance (as explained below) is only about 8-10% when we move from a square to a linear array.

5.2 Clock-Rate Reduction

Another useful way of looking at the coprocessor is to compute the ratio of clock speeds for comparable performance to a commercial high-speed RISC microprocessor (say with an arithmetic pipeline). Hopefully, by exploiting the structure of the algorithm, the coprocessor should be able to deliver the same performance (or throughput) as a RISC processor for much lower clock speeds. Therefore,

$$\begin{aligned} T_{compl} \times \frac{1}{Clock_{PA}} &= N^3 \times \frac{1}{Clock_{RISC}} \\ \Longrightarrow \frac{Clock_{RISC}}{Clock_{PA}} &= \frac{N^3}{T_{compl}} \end{aligned} \qquad (7)$$

We assume that there are no pipeline stalls, no data-access penalties (cache misses) and no loop-control overheads in the RISC processor. Hence, the ratio obtained above represents a lower bound on the possible reduction in clock rates.

A reduced clock rate is desirable for several reasons. First and most important, the yield of the coprocessor will be significantly higher if it is designed for a low clock rate. For instance, the yield would improve significantly, with most of the chips on the wafer acceptable, if the chip is designed for 1-Mhz instead of 50-Mhz clock rate. In addition, power dissipation is lower at lower clock rate, which leads to better integration and lower packaging costs. Hence, reduced clock rates will lead to significantly lower system cost. This is intuitively in agreement with existing high-speed processors where there is a steep (possibly exponential) variation in cost (and design time) with clock speed.

5.3 Evaluation

We present a sample of results in exploring the space of architectures for 2-D-mesh PAs (without wraparound) and linear PAs with only 2 peripheral (I/O) processors.

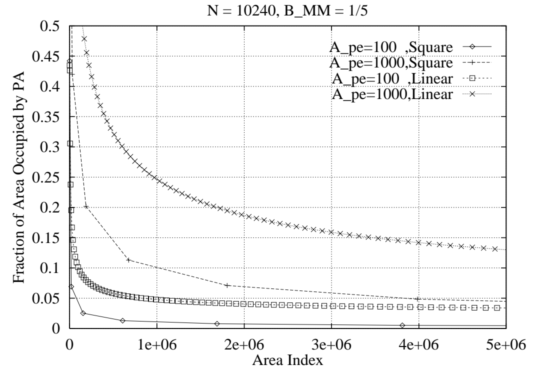

Figure 4: Variation of area occupied by PA as function of total area.

Figure 4 shows the variation of the fraction of the total area occupied by the PA as we increase the total silicon area of the coprocessor. The size of the AU in Eq. 4 is computed using p from Eq. 6, which is chosen to overlap memory fetches completely with computations. The x-axis is the area of the chip in memory words. Thus, half a million words corresponds to an area equivalent to 16 million bits of storage, assuming each word is 4 bytes. This is approximately where the limit of current technology is today. A_{pe} refers to the cost of a PE again in terms of memory words. Thus, $A_{pe} = 100$ means each PE takes an area equivalent to 100 words of memory. Hence, for 500 K words of total area, only 4% of the total area is occupied by processors for a linear-array configuration with $A_{pe} = 100$. This shows that most of the chip area is used by the AU if we design the chip for optimal balance, where memory latency is fully masked. Moreover, for the same total area, the linear array has more area devoted to processors. The effect of increased area of a PE is to lift the entire plot upwards both for linear and square arrays. Note that although this figure is for multiplying matrices with $N = 10240$ and $B_{MM} = 1/5$, the same effect is observed for other problem sizes and bandwidths.

Figure 5 shows the reduction in clock rates possible for $N = 10240$ with increasing chip sizes. The x-axis is the area index measured in memory words. Again, the system is designed at the balance point to mask memory latency. Figure 5 shows that for about 500 K words of silicon area, the clock rate can be reduced by a factor of about 175 for a square array and 160 for a linear array to get the same execution time as a pipelined RISC processor. Therefore, for a

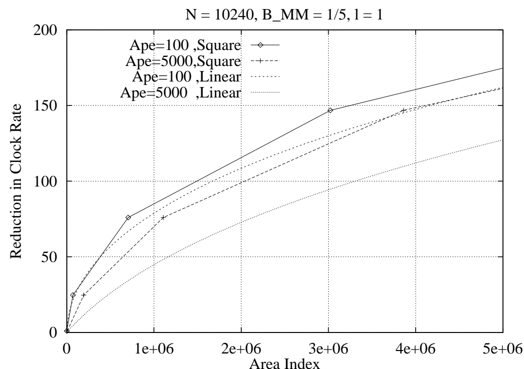

Figure 5: Reduction in clock rates for N=10240 with increasing chip sizes.

10240×10240 problem with a latency of 5 cycles per memory word and clock rate of 1 Mhz, we will get a performance equivalent to that of a RISC processor running at 160Mhz (or 6.25 nanoseconds cycle time which is closer to that of supercomputers than that of current RISC processors). For a 66Mhz clock rate (15 ns), we can clock the coprocessor at 400 Khz for the same performance (which is about 160 MFLOPS), or at 1 Mhz and get a 2.5 times speedup. The final speed can be chosen from a variety of attractive alternatives depending on the objective of the designer. As shown in the figure, the clock-rate reduction saturates as area index increases, indicating diminishing returns from parallelism. The degradation in performance in using a linear array to a square array is at worst 8% for $A_{pe} = 100$ and 20% for $A_{pe} = 5000$. Therefore, the linear-array configuration is an attractive choice for matrix multiplication and other UREs. Its advantages are constant I/O bandwidth, modular expandability, and efficient VLSI layout.

6 Conclusions

We have presented an approach for designing a coprocessor tailored for executing loop computations described as uniform recurrences. Our approach uses provably optimal methods that result in high utilization and effective data-locality management. We show (i) that a scalable, linear array of PEs with constant bandwidth achieves good performance, and (ii) that we can design the coprocessor with significantly lower clock rates than an existing RISC processor with comparable performance, or get higher performance with comparable clock rates.

Acknowledgements

We would like to thank Prof. M. C. Loui for many fruitful discussions which improved the quality and presentation of this paper greatly.

References

[1] S. Borkar, "iWarp : An integrated solution to high-speed parallel computing," *Proceedings of Supercomputing*, (Florida), pp. 330–339, Nov. 1988.

[2] "Special issue on systolic arrays," *IEEE Computer*, vol. 20, July 1987.

[3] K. W. Przytula, "Medium grain parallel architecture for image and signal processing," in *Parallel Architectures and Algorithms for Image Understanding* (V. K. P. Kumar, ed.), pp. 95–119, Academic Press, 1991.

[4] J. H. Moreno and M. E. Figueroa, "A decoupled access/execute processor for matrix algorithms: architecture and programming," *Application Specific Array Processors*, pp. 281–295, IEEE Computer Society Press, 1991.

[5] D. I. Moldovan and J. A. B. Fortes, "Partitioning and mapping algorithms into fixed size systolic arrays," *IEEE Transactions on Computers*, vol. C-35, pp. 1–12, Jan. 1986.

[6] Z. Chen and W. Shang, "On uniformization of affine dependence algorithms," *Proc. Fourth Symposium on Parallel and Distributed Processing*, p-p. 128–137, Dec. 1992.

[7] U. Banerjee, *Dependence Analysis for Supercomputing.* Kluwer Academic, 1988.

[8] J.-W. Hong and H. T. Kung, "The I/O complexity: The red blue pebble game," *Proceedings of 13th annual ACM Symposium on Theory of Computing*, pp. 326–333, May 1981.

[9] K. N. Ganapathy and B. W. Wah, "Synthesizing optimal lower dimensional processor arrays," *Proceedings of International Conference on Parallel Processing*, CRC Press, Aug. 1992.