# Overcoming Memory Latency And Enabling Parallelism With The Greedy CAM Architecture

Ray Bittner

January 31, 2007

MSR-TR-2007-10

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

# Abstract

Computing today is inexorably headed towards increased parallelism. Using a hybrid dataflow approach to the problem, the Greedy CAM architecture addresses issues of programmability and some of the realities of today's hardware, such as the clock rate plateau and the curse of memory latency. Greedy CAM employs a more generalized use of tags than previously seen in tagged token architectures in that the tags are related functionally at the inputs to each of the computational kernels. A Content Addressable Memory works with the functional tag relationships in order to deterministically prefetch operands to feed a pipelined computing engine. Beyond simple contiguous or stride based operand accesses, the CAM enables truly random memory access patterns and conditional execution that are friendly to a parallel, pipelined, machine. Functional tag relationships and the CAM are used together to provide main memory, synchronization and RAM abstraction. Kernels are run on the machine using a coarse grained dataflow control scheme that can dynamically expand or contract to fit the available parallel hardware at runtime. Lastly, the architecture lends itself to several implementations, including reconfigurable hardware or a software approach using a many core like architecture.


## I. Introduction

The Greedy CAM architecture is a hybrid dataflow computer that makes use of tagged data relations to build sets of operands in a streaming manner for consumption by pipelined processing blocks. A conceptual view of the architecture is shown in Fig. 4 below. The CAM is employed in multiple roles: as data memory, as a synchronization mechanism and as a means of abstracting normal RAM accesses. The architecture is referred to as greedy due to the greedy fashion in which operands may be fetched out of order from memory as well as to the potentially greedy scheduling scheme that can be used to execute the architecture's many Kernels.

The design of the Greedy CAM architecture has been created against the backdrop of the state of the art in computational hardware, with all of its warts. Most apparent today is that the Moore's Law trend of clock rate speedup has slowed down [1,2], and as a result the increase of clock rate alone cannot be counted on to drive continued speedups in hardware. At the same time, logic density is continuing to increase, at least up to a point

[3]. The answer seems to be some sort of parallelism, which Greedy CAM offers in several forms. In terms of spatial parallelism, Greedy CAM offers a change in the programming paradigm that makes it possible to write a single program that can dynamically expand and contract to fit the available execution hardware, whether that hardware is constrained by unknown system size, or other processes running in the same system.

Temporally, Greedy CAM enables pipelined execution by making data dependencies explicit in the programming model. This eliminates the problem of pulling apart a program to infer the intended computational trees which are often interrelated and/or self-referential [4]. Further, Greedy CAM supports the notion that producers and consumers may be directly connected within the computational hardware, so that static scheduling may optimize the performance of the pipeline.

Memory latency has become an increasing burden on modern computational systems. This comes mainly from two sources. First, today there is roughly a 5x differential between the rate that data items are clocked out of a memory chip and the rate that they may be consumed by a processing chip. This differential is related to many factors, including silicon process technologies, chip packaging techniques and PCB technology. Secondly, the on-chip storage and retrieval mechanisms of DRAMs are limited by the switching speeds of relatively highly capacitive lines.

Architecturally speaking, many techniques have been proposed to overcome the collective latency of main memory. Caches [5], dynamic instruction scheduling [6,7], branch prediction [8,9], speculative execution [10,11], thread based speculation [12, 13] and other techniques have been proposed. Generally, these ideas provide gains by their ability to predict data accesses and prefetch the correct data before the flow of execution actually requires it. Rather than trying to divine data accesses through these techniques, the Greedy CAM architecture combats the memory latency problem by providing the hardware with explicit indications as to which data values will be needed next, removing the need for prediction logic.

Lastly, DRAM designers have bestowed hardware architects with one boon that is often underutilized. When accessing most DRAMs, including the familiar DDR2 variety, the physical address is split roughly in half, where the upper bits are used to open a specific row of the rectangular bit array and the lower bits are used to access a subset of that row. Once a given row is opened, access to any given byte within that row is very fast, and, with

pipelining, a continuous stream of values can be read from it in DDR fashion. If the row address must be changed; however, the RAM architecture demands a break in the data stream of approximately 12 memory clock cycles while the bits for the new row are latched into internal registers [14]. Assuming a DRAM with a memory clock of 400MHz and a processor running at 3GHz, 12 memory clocks correspond to at least 90 processor clocks where that particular pipeline, or at least thread, is completely stalled. Thus, it is advantageous to ensure that memory accesses demand as few changes in the row address as possible, and a memory hierarchy that took explicit advantage of this fact could benefit.

This optimization could be made through some combination of careful programming, the virtual paging scheme of the operating system, and hardware caches. Optimizations of this kind are generally not the first consideration of the programmer or the operating system as it allocates physical pages of RAM. However, it must be realized that today's Random Access Memory performs better when the memory accesses are not truly random. It should be possible to improve the performance of the programming model if the machine incorporates and encourages localized external memory operations so that programs require fewer row address changes. Because Greedy CAM abstracts the memory system behind a tagged data system, a memory system design has been devised to do just that.

The unique contributions of the Greedy CAM architecture that will be discussed in this paper are:

- A more generalized use and definition of data tags than has previously been seen.
- A functional tag relationship mechanism that allows for the intelligent prefetch of data under programmer control.
- A control system that is distributed in nature, allowing scaling of the system and greater spatial parallelism.
- Mechanisms for conditional execution and looping using an associative memory and functional tag relationships.

# II. Functional Tag Relationships

Consider the familiar code:

```
int i, A[10], B[10], C[10];
for( i = 0; i < 10; i++ )
    C[i] = A[i] + B[i];
```

The intent is instantly recognizable by anyone with a modicum of programming ability. However, its interpretation from the machine's point of view is over specified. While the result does not require the sums to be calculated in a specific order, the semantics of the language, and the underlying machine code, dictate one. It is possible for various techniques to be used to recognize this type of situation in software or hardware, but quite often pointer aliasing, etc, makes this quite difficult. Further, it can be difficult to determine whether the implied sequential execution paradigm is critical to the correct evaluation of the code.

The origins of this style of programming are rooted in Random Access Memory addressing. We supply an address and the RAM returns the value at that address. The address and the value at that address are generally unrelated to one another. With the vast number of transistors that can be implemented using today's technology, it may be time to reconsider this memory organization in favor of something that can help address the processing needs of today. If the data labels itself through tags, and allows access based on that mechanism, the memory is abstracted in a way that distributes the processing burden. The tags can be used as a matching mechanism to associate similar data elements with one another for processing, rather than using the processor to compute the addresses of related items and then using those addresses to fetch the data items themselves. With tags, the processor needs only to specify the relationship that paired data items should have with one another, and the memory can do the dirty work of finding pairs of data that satisfy that relationship. Another way of writing the code above might be:

```
int i, A[10], B[10], C[10];
C[i] = A[i] + B[i];
```

Here it is understood that all possible index values that satisfy the given relation will be processed. In this representation, the order that the indices are processed is left up to the machine. Moreover, the inter-relationship between elements is not obscured by the inherent sequential execution paradigm of the standard C representation. The index values become active parts of the expression, just as much as the operation that is being performed. Other possible expressions could include:

```
C[i] = A[9-i] * B[i];
C[3/i] = A[i] + B[2*i];
…
```

This could be called a functional relationship between program operands, rather than an imperative one. Further, a type of content addressable memory is suggested, rather than one based on random access of addresses. The evaluation of the expression then becomes an exercise of searching the CAM for matches to computed indices as dictated by the expression. The full set of operands needed to compute the expression for a given value of i is called an Operand Set. Note that it is not necessary for there to be a match for every given operand. It is permissible for there to be an A[3] when no matching B[3] exists. In that case, A[3] would be fetched, but when the hardware finds no matching B[3] it would merely discard it as an Operand Set candidate and move on to the next A[i].

The Operand Set representation gives the hardware a more explicit picture of the possibilities for parallel execution of the operation across the elements. Meaning that C[0] can be evaluated concurrently with C[1], either through pipelining or through spatial parallelism. Another benefit is the use of wildcard indices to give an a-priori knowledge of the memory accesses that will be performed. When the A[i] expression is encountered, the hardware can issue a single memory request for all elements of the array A. As these are returned, the appropriate memory requests for elements of array B can be issued based on the functional relationship of the indices. In this way, the memory access process becomes a streaming operation while still allowing for the possibility of interesting access patterns. Memory latency can be hidden (after startup costs) by prefetching the A[i]'s in this case and issuing requests for corresponding B[i]'s as the A[i]'s arrive.

## III. Functional Tag Example

More advanced tagged relations are possible, as illustrated in Fig. 1. Here, the partial products for a 3x3 matrix multiplication are being staged and the $\perp$ symbols represent empty matrix elements. The expression for generating the partial products needed for the multiplication is shown below the matrices, and

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix} * \begin{bmatrix} \perp & b_{01} & b_{02} \\ b_{10} & b_{11} & \perp \\ \perp & b_{21} & b_{22} \end{bmatrix}$$

$$P[i,j,k] = A[i,j] * B[j,k]$$

| | | | | | |
|---|---|---|---|---|---|
| $a_{00}$ | $b_{01}$ | $a_{10}$ | $b_{01}$ | $a_{20}$ | $b_{01}$ |
| $a_{00}$ | $b_{02}$ | $a_{10}$ | $b_{02}$ | $a_{20}$ | $b_{02}$ |
| $a_{01}$ | $b_{10}$ | $a_{11}$ | $b_{10}$ | $a_{21}$ | $b_{10}$ |
| $a_{01}$ | $b_{11}$ | $a_{11}$ | $b_{11}$ | $a_{21}$ | $b_{11}$ |
| $a_{02}$ | $b_{21}$ | $a_{12}$ | $b_{21}$ | $a_{22}$ | $b_{21}$ |
| $a_{02}$ | $b_{22}$ | $a_{12}$ | $b_{22}$ | $a_{22}$ | $b_{22}$ |

**Fig. 1 Matrix Multiplication**

the input Operand Sets that would be retrieved are shown below that. The B matrix is sparse, which accounts for only 18 Operand Sets being assembled rather than the expected 27. After the partial products are computed, another expression could then be used to associate the appropriate partial products and perform the sum operations.

The tag manipulations for the partial product generation are shown in Fig. 2. The first operand is retrieved by setting the upper bits of the tag to a constant used to reference the A matrix, with two sub-fields i and j that are both wildcards. This will result in a request for all nine elements of the matrix A. As these are returned, the j sub-field for each is forwarded to the second operand to select elements from the B matrix that match. The tag for the second operand is formed using a constant that references all possible elements of the B matrix, but is then qualified further using the j sub-field bits from the first operand as constants. The k sub-field of the second operand again contains wildcard bits. The Operand Sets may be assembled in any order, subject to the functional relationship of the input tag specification; allowing maximum flexibility for the hardware to make optimizations. Lastly, the output tag for each of the partial products is formed using a constant P along with the i, j and k bits from each of the two input tags. Concatenating the three fields i, j and k is not a problem since the Greedy CAM architecture supports variable width tags.

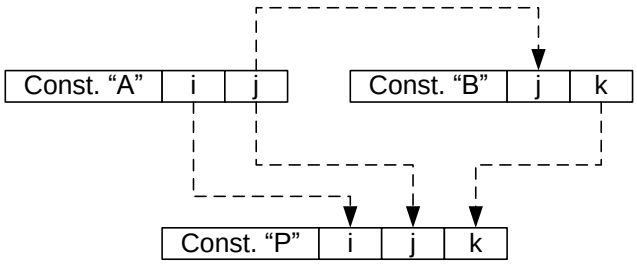



**Fig. 2 Matrix Multiplication Tag Relationships**

Functional tag relationships and Operand Sets alone can serve to address all four of the problems discussed in the introduction. Memory accesses can be predicted as described. Spatial and temporal parallelisms are both supported since computation can be performed as completed Operand Sets are assembled. By abstracting the memory into a CAM, and requiring the software deal with tag references alone, the responsibility for managing physical memory addresses falls to the hardware, which can make efforts to localize the memory access pattern to fall within RAM pages.

Lastly, a matrix multiplication example is trivial to implement with a vector computer with the same results. The same is also true of macro dataflow machines. The power of the Greedy CAM architecture comes from the

non-linear memory access patterns and logic that the CAM enables. The sparse matrix shown above is one example of that, but bear in mind that the bit manipulations and computations between tags and/or data can theoretically produce any memory access pattern. Further, the fact that it is legal for some of those operands to be missing in a way that is seamless to the execution allows for more interesting possibilities such as using the CAM for conditional execution as covered below.

# IV. Execution Flow In Brief

Greedy CAM's pipelining possibilities may be improved by growing the size of the expressions, as well as by allowing for the possibility of multiple outputs in the event that they share a common sub-expression. For even greater opportunities for spatial parallelism, it is beneficial to consider executing several independent expressions concurrently. Again, it would be helpful if the programming paradigm provided explicit cues to the hardware as to when a strict execution order is required and when it is not.

To this end, the program is divided into individual Kernels with explicit dependence links between them, called Signals. This could be done by the programmer, the compiler, or a combination of the two. Each Kernel consists of one or more expressions of the type that has been discussed so far. Fig. 3 shows an example graph.

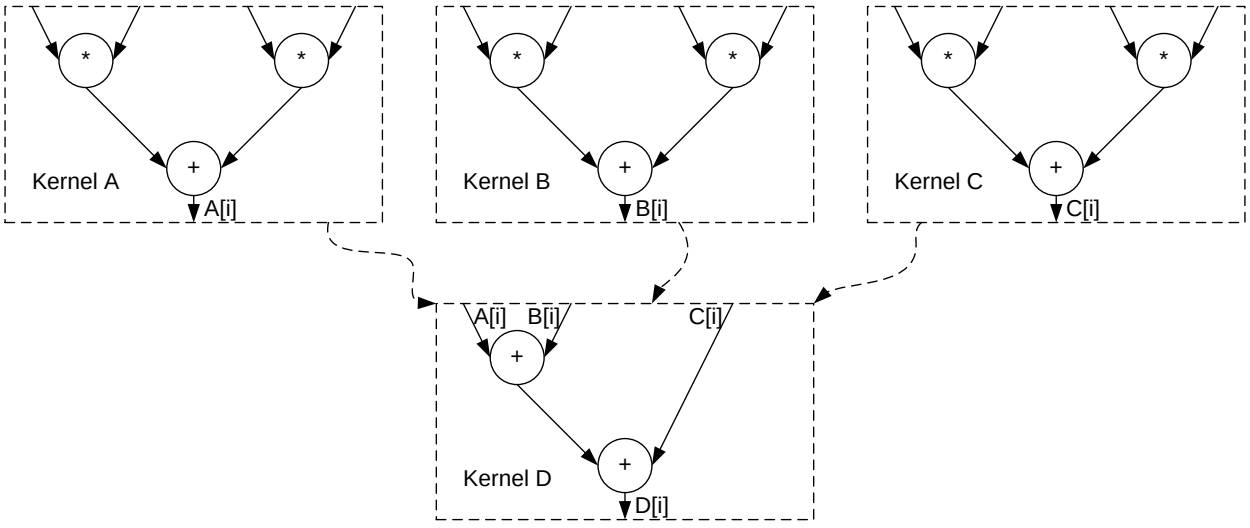


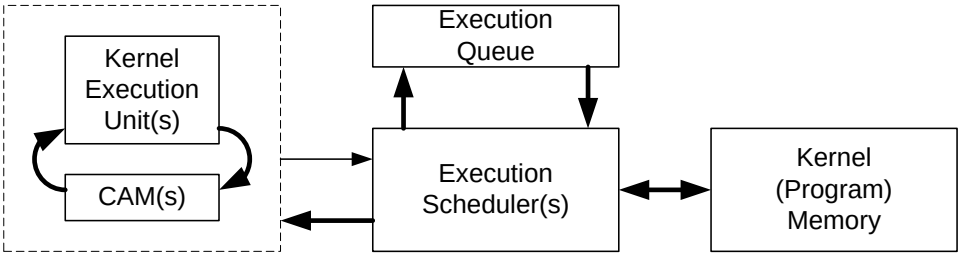

**Fig. 3 Parallel Kernel Organization**

The dataflow structure is obvious, with all of the benefits that entails. The arcs shown between Kernels represent the execution order required by the program and so the hardware is able to easily find Kernels that can be scheduled for execution in parallel. This representation allows for the possibility of dynamically growing and shrinking the number of active Kernels to match the available hardware and parallelism in the algorithm at runtime, giving the program an automated scaling ability.

The Signals also happen to reflect the flow of data as shown, but this need not be the case. In fact, the input/output relationship of data between Kernels is contained in the output and input tag definitions themselves, and is not derived from the dependence arcs. For example, Kernel A produces values that are tagged as A[0], A[1], A[2], etc, Kernel B produces B[0], B[1], B[2], etc, and so on. Kernels A, B and C may be executed in any order, as long as they have all executed before Kernel D is scheduled. The input tag specifications for Kernel D are A[i], B[i], C[i], to match the tags that were emitted from the previous Kernels. In this case, the Signals could be inferred by a compiler, though there are more interesting uses where they cannot be easily inferred.

Partitioning the overall graph into Kernels is an optimization process that must consider many different factors. There may be hardware limitations as to the maximum size of a given Kernel, for example. More than this, the selection of Kernel boundaries will have an impact on the performance of the machine. For example, if it is possible to combine multiple expressions containing a common sub graph into a single Kernel, then the operands that are used to compute that sub graph need only be fetched once. In fact, it would often be beneficial to include the largest possible sub graph within the same Kernel since each data path that is broken could represent a trip to memory. Alternatively, one could optimize the Kernel boundaries to cut the minimum number of data paths. There are many possible strategies and solutions.

# V. Greedy CAM Execution Engine




1. Retrieve Next Kernel
2. Run Kernel To Completion
3. Update Dependent Kernels With Signal Status
4. Place Newly Readied Kernels Into The Execution Queue

**Fig. 4 Greedy CAM Execution Engine**

Fig. 4 shows the basic execution loop, which is equivalent to the fetch-execute cycle of a control driven processor. The programming information necessary to reconstruct each Kernel is stored in the Kernel Program Memory shown to the right. At the beginning of program execution, the IDs of one or more initialization Kernels are placed in the Execution Queue. From that point on, the Controller follows the four steps shown. A Kernel to be run is selected from the Execution Queue and is sent to an available Kernel Execution Unit (KEU) for execution. The Kernel runs to completion, and as it does so it sends Signals back to the Controller indicating the success or failure of various operations. The Controller uses these Signals to set flags in Kernels that are dependent upon the current Kernel. Dependent Kernels that are found to be ready for execution are added to the Execution Queue for later scheduling into a Kernel Execution Unit.

Current simulations of the architecture use just one controller, which is capable of efficiently managing quite a few Kernels since the amount of control and Kernel information needed is small compared to the amount of data that each Kernel will process. However, if the number of Kernels is large or the amount of data per Kernel is small, it is possible for multiple Controllers to execute in parallel. In fact, it is possible to use a CAM to store the Signal state of each Kernel, and then consider the set of activation Signals for a Kernel as an Operand Set. As each Signal for a given Kernel is stored as tagged data in the CAM, it could form an Operand Set that when completed, will trigger the addition of that Kernel to the Execution Queue.

Once a Kernel has been placed in the Execution Queue, there are many possible scheduling strategies that could be employed. To date, experiments have only been run with a simple FIFO strategy; running Kernels in the order that they become available for execution. However, it is easy to envision scheduling based on different criteria such as depth first, breadth first, some concept of data availability that could aid a data caching system, or the number of dependent Kernels, etc.

# VI. If/Then Conditionals

Conditionals and branching are often the weak points of a pipelined architecture. Consider the vector If/Then:

```
if (A[i] < B[i])
    C[i] = D[i] + E[i];
else
    C[i] = F[i] – G[i];
```

It would be beneficial to organize the execution of this structure so that for each index i, only the side of the conditional required is executed. Doing so maintains a fully utilized pipeline for each side of the conditional, and also eliminates potentially wasteful memory accesses. To accomplish this, Greedy CAM uses a variant of the $\Phi^{-1}$ instruction as is used in WaveScalar [24] and other architectures. Fig. 5 shows the Greedy CAM implementation of the same structure. Here, the CAM is used together with the Operand Set concept to filter inputs for processing. As shown, the data values D[i], E[i], F[i] and G[i] have been previously written into the CAM as they were produced. Three Kernels function as the If/Then structure. First, a selector Kernel performs the comparison between A[i] and B[i]. If the outcome indicates that the left branch of the If should be taken, it writes a SL[i] seed tag back to the CAM. If the outcome indicates the right side, a SR[i] seed tag is written to the CAM. After that, two additional Kernels are run representing the code in the left and right sides of the If statement.

The left Kernel has an input Operand Set defined by SL[i], D[i] and E[i]. Hence, it will only select the operands for which seed tags have been written from the CAM, and C[i] outputs are produced for those values (in this case C[1] and C[2]). Similarly, the right side Kernel has an input Operand Set defined as SR[i], F[i] and G[i], which causes it to pick up seed tags SR[0] and SR[3] and produce outputs C[0] and C[3].

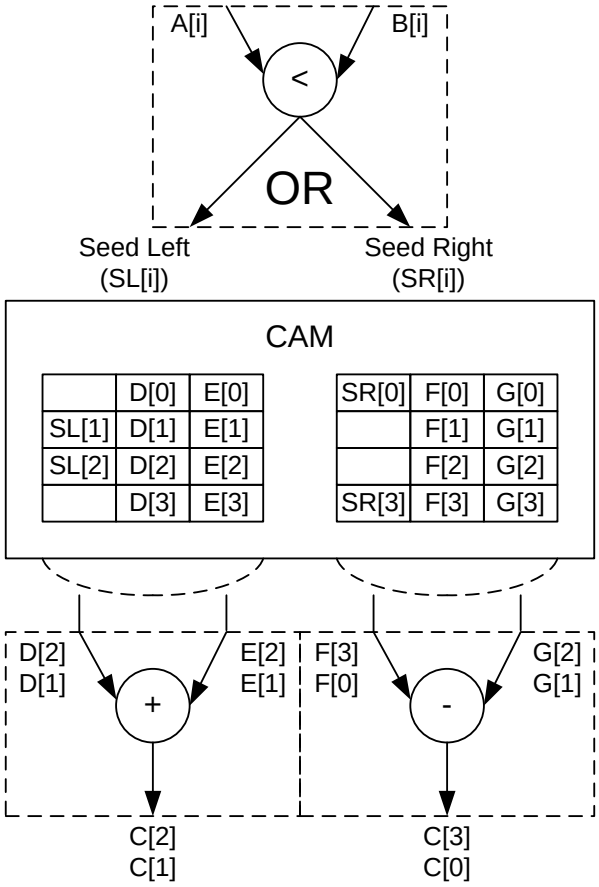



**Fig. 5 Greedy CAM If/Then**

In some sense, this is an analog of predicated instructions on a RISC machine, except that in the Greedy CAM case, the predication happens in tag space rather than in the instruction pipeline. Taking it a step further, a form of short circuit evaluation may be implemented when the discriminating values (A[i], B[i]) are available before the computational values (D[i], E[i], F[i] and G[i]). If this is the case, then the seed tag mechanism can be used to prevent the creation of the computational values at the point where they are created, thus saving more computation.

In terms of memory accesses, a memory request will still be made for the seed tag on the failed side of the If/Then. However, this would be part of a wildcard tag specification of the form SL[i] or SR[i]. With a well designed CAM, the absence of a tag within the wildcard range will add little or no delay to the retrieval of tags that are present in that range. Thus, the memory accesses for the failed side of the If/Then can be effectively eliminated.

The entire process is pipelined and lends itself to large vectors of data. Further, it is possible for the left and right Kernels to run simultaneously since they are guaranteed to be mutually exclusive. Lastly, it is possible to

conditionally Signal the execution of entire Kernels rather than simply generating seed tags. This allows large sub-graphs to be dynamically swapped in and out at run-time as appropriate.

## VII. While Loops

While loops can also make good use of the CAM and can use a slightly more advanced application of the same mechanism. Root finding for a vector of values is a good example:

```
while (abs(f(X[i]) – Target) > Delta)
      X[i] = Update(X[i]);
```

Fig. 6 shows one possible implementation consisting of three Kernels. The overall structure is shown, with the three Kernels boxed in dotted lines and labeled Generate, Update/Test and Next. The Generate Kernel runs only once, and is responsible for creating the initial X[i] values, along with a Seed Continue tag (SC[i]) that is associated with each. Depending on the program, this needn't be a separate Kernel since the generation of the SC[i] tags needs only be done at the point where the X[i]'s are produced.

The Update/Test Kernel runs next, forming an Operand Set from X[i] and SC[i]. This will become the barrier that will prevent completed X[i]'s from re-entering the loop. The first operation in this Kernel is to generate a new value of X[i], which is written back to the CAM and also forwarded for testing against the termination condition. If the termination condition has not been reached, an SC[i] tag is written to the CAM. As the Kernel continues to run, or when it is run again, the SC[i] and X[i] tags will be read again for the next iteration. If the termination condition has been reached, no SC[i] tag is written, but a Seed Stop (SS[i]) tag is written to the CAM instead. Since no SC[i] tag was written out upon termination, that X[i] value will effectively fall out of the loop since it will no longer form a complete Operand Set for the Update/Test Kernel. It will, however, form a complete Operand Set for the Next Kernel using SS[i] and X[i], allowing processing to continue downstream.

Each X[i] value flows through the loop independently from all the others. Since the exact number of iterations needed for each X[i] is unknown, this independence is useful because it allows the Update/Test Kernel to keep a full pipeline of values for as long as possible. For example, X[0] and X[2] may continue processing as X[1] and X[3] fall out of the loop, while at the same time the Kernel can pick up X[4] and X[11] as new inputs to fill those holes in the pipeline. This should lead to a highly efficient use of the pipelined sub-graph.

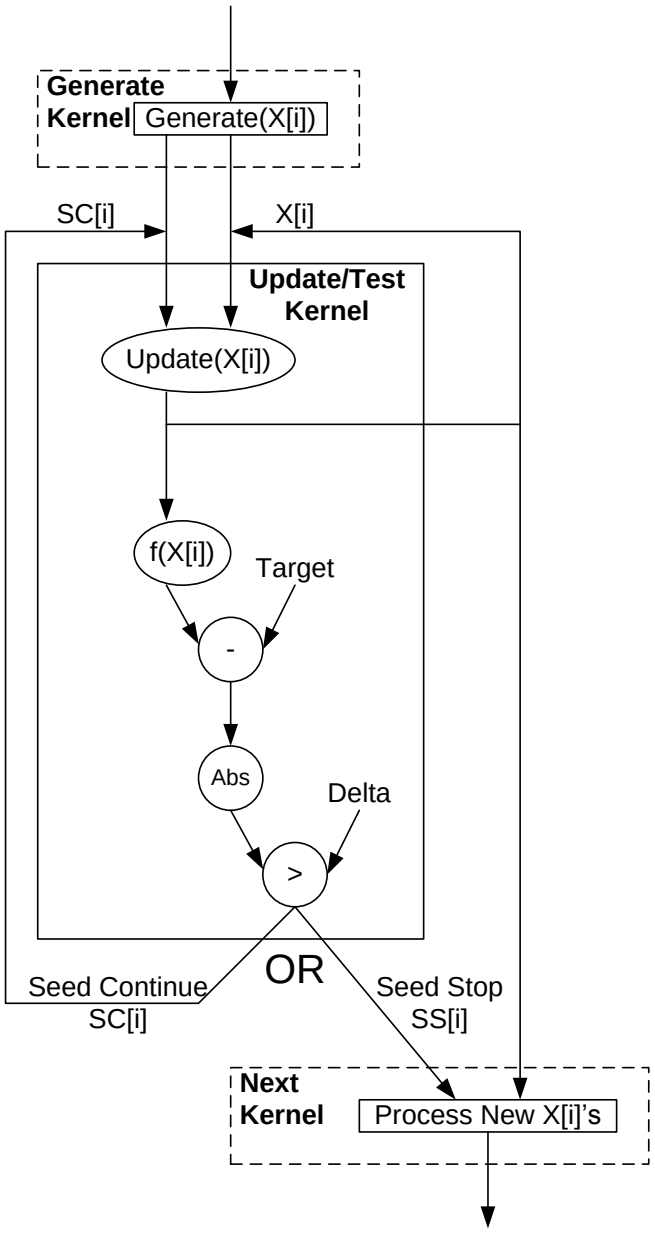



**Fig. 6 While Loop Example**

## VIII. CAM Design

Much of the performance and feasibility of the Greedy CAM architecture depends on the design of the CAM itself. Unlike the typical array of memory cells and comparators, the CAM for this architecture is intended to be implemented using commercial DRAMs so that it can be both large and inexpensive. The hardware maintains data structures in the RAM that allow it to quickly arrive at the correct tag values in response to a tag query. Bear in mind that the query specification may contain wildcard bits in arbitrary positions, which takes it out of the realm of a ranged query. Fig. 7 shows the TRIE based structure that is employed.

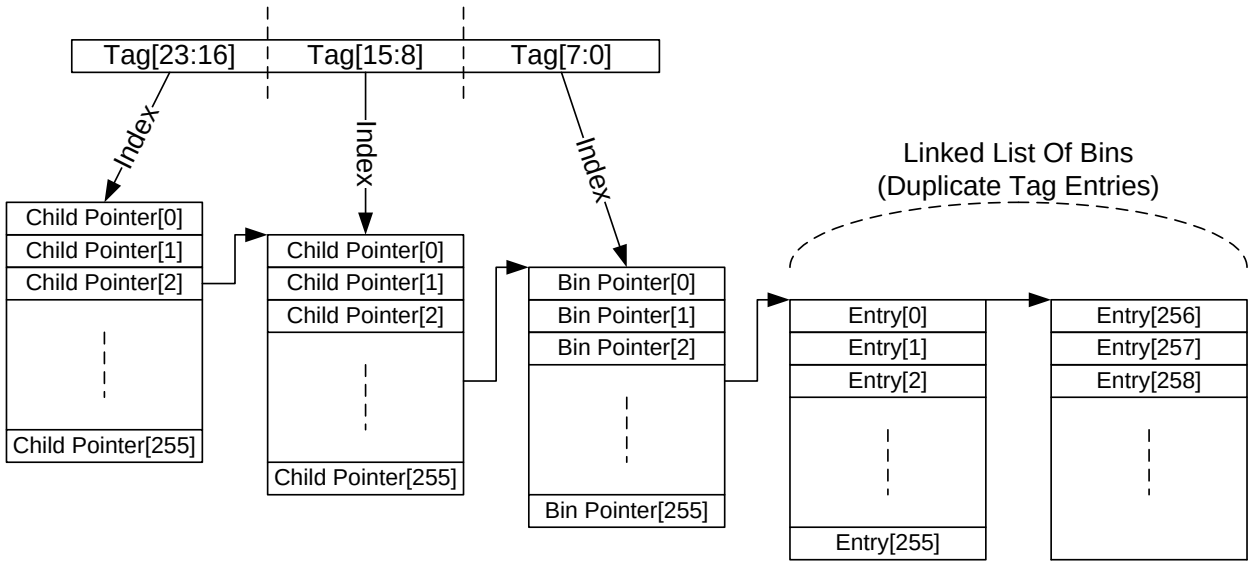



**Fig. 7 External CAM Organization**

The variable parts of the tag are subdivided into bit fields; shown here as eight bits wide. Each field is used as an index into a table that gives a pointer to the table of pointers used for the next field. Each entry of the table for the last field contains a pointer to a linked list of bins. Each entry in a given bin has been assigned the identical tag for identification purposes and will be returned as a duplicate result for any query. The ability to support duplicates adds efficiency to some algorithms. Accesses to the entire data structure can be pipelined as long as the normal read/write rules are observed. Obviously, the exact width of the bit fields is subject to optimization based on the computation, memory utilization and access time.

Note that each of the bin entries can be guaranteed to reside in consecutive physical RAM addresses by the hardware, which supports the fast burst mode of SDRAMs. Likewise, the hardware manager has a great deal of control of how and where memory is allocated in order to minimize RAM page misses. Also, the data structure allows for early termination of non-existent tags by providing the equivalent of a null pointer in the table at the earliest level possible. This saves memory overhead for the representation as well as memory bandwidth.

It would not be practical for the hardware to page in the entire contents of each of these tables for every query due to memory bandwidth constraints. Instead, each table would actually have an associated smaller array of presence bits, with one presence bit for each pointer entry. The presence bits for the tables shown above might consume 256 bits or 32 bytes which could be scanned in a single burst read from an SDRAM. A caching scheme

could also be used with these tables to reduce the frequency of reads from external RAM. The process of enacting a query would consist of loading the presence bits for the table of interest, comparing the constant bits of the query for that subfield against the presence bits of the table and using carry chain-like logic along with the wildcard bits to choose the next pointer index to pursue.

Lastly, the data structure in Fig. 7 represents the handling of all of the variable bits for a given tag. The upper constant bits (such as "Const. 'A'" in the examples above), can be used to greater effect by allowing the programmer or compiler to allocate a physically separate data structure for each constant prefix. This allows the efficient representation of variable length tags by the addition of more bit fields to longer tags and the corresponding growth of the data structure, while also allowing for shorter tags and a smaller data structure where applicable. Further, the upper constant bits may also be used to place each data structure into a physically separate CAM unit under software control. Each of these CAMs could have separate memories and controllers, allowing for parallel accesses. This allows for the possibility of synergy between the compiler and the hardware architecture since the physical location of some tables may affect the potential parallelism of the query process.

# IX. Discussion

The Greedy CAM architecture is designed for applications with large amounts of data and a large amount of spatial and/or temporal parallelism. Much of the novelty is derived from the multiple uses of the CAM itself. It functions as a memory, a synchronization mechanism and as a means of abstracting the normal RAM access patterns by allowing more complex queries than simple address based retrieval. This abstraction is a way of distributing some of the computational burden across what can be a system of one or more CAM(s). Since the computations themselves can occur out of order subject to relatively gross constraints, it is possible to create multiple parallel CAMs that may all play a part in serving a particular tag request. The result is a system that can service the tag specifications put forth by the Kernels using long latency, high bandwidth, components.

This puts a large demand on the design of the CAM itself since it functions as the system's primary data storage. The design put forth makes use of commercial DRAM technology while matching the characteristics of the rest of the system; functioning in a pipelined manner where longer latency can be tolerated as long as high

bandwidth is maintained. There are commercial chips available for implementing CAMs in normal DRAM, mainly for use in network routing applications. However, they do not satisfy all of the desires of the Greedy CAM architecture, either in size, or in the fact that they use fixed width tags.

There is also a related question as to what hardware should be used to execute the Kernels. There are many potential answers to this question. One implementation would use many small control driven cores, reflecting the trend of current commercial hardware. In that incarnation, Operand Sets could be dealt out to several cores each of which was executing identical code representing the Kernel, since the Operand Sets themselves are guaranteed to be independent of one another. It is easy to imagine that the number of cores dedicated to executing a given Kernel could be dynamically adjusted to the amount of data processed by that Kernel, or to the urgency with which it must be completed, or to the current work load of the system.

Another implementation of the Kernel hardware would be to use reconfigurable hardware blocks [15]. Since the computational units within these could be easily pipelined and connected in a myriad of ways, it would seem to be a natural fit. Due to the pipelined nature of such blocks, it is likely that fewer of them would be needed than if the control driven core approach were taken. It may also be possible to specialize them to a greater degree than control driven cores would allow. Each Kernel would then be partitioned from the overall graph subject to the size limitations of the reconfigurable hardware block in addition to the other considerations mentioned above.

Lastly, schemes have been devised for function calls and recursion, but a full description of these would be a lengthy discussion. In short, function calls may be implemented by passing the arguments into a Kernel as part of a normal Operand Set, and then adding one or more additional arguments to the Operand Set to indicate the constants to be used for the corresponding output tags. Recursion is implemented using a similar mechanism, except that the recursive routine is split at the point(s) of the recursive call(s), and a subfield of the output tag of each call is incremented for each recursion. A cleanup sweep is then performed across all of the partially evaluated recursive calls to complete their evaluations in reverse order.

# X. Related Work

In terms of the programming model, Greedy CAM most closely resembles Linda [16,17]. Many of the differences lie in the mechanics of implementation. Where Linda strives to be a portable parallel programming language, Greedy CAM attempts to optimize the semantics to lend itself to an efficient hardware implementation. Linda's tuples could be placed into Greedy CAM's variable length tags, for example. A Greedy CAM tag is just a string of bits as far as the hardware is concerned though; whereas, a Linda tuple contains more "type" information and may in fact require several indirections through memory to resolve. Greedy CAM also makes dependencies between Kernels explicit, so that the hardware has an easier time determining the producer/consumer relationships between Kernels. Another difference is the way that contention for deletions of operands is handled. Linda proposes different broadcast protocols for guaranteeing that only one consumer is able to fetch a particular tuple. For the sake of efficiency, the current implementation of Greedy CAM instead demands that the compiler or programmer partition the tag space such that when deletions are required, only one consumer is allowed to exist. This may appear to be a loss of parallelism; however, the tag space can be pre-partitioned by the compiler on power of two boundaries using the wildcard mechanism, and Kernels can be replicated in an a-priori fashion for that case. There are several other differences in execution, design and programming.

In terms of hardware architectures, Greedy CAM has similarities to many proposed machines; however, the Operand Set matching mechanism is distinct among them. Tagged token dataflow machines [18], such as TTDA [19], the Manchester machine [20], Monsoon [21], and others also store each data item with a tag by which it is referenced. These machines all use an inherently "push" mechanic where the arrival of a new operand at a node causes the node to scan for that operand's mate(s) so that processing may continue. If the mate(s) cannot be found, that operand must be buffered up by the node and is then the subject of future scans when other operands arrive. The Greedy CAM architecture, on the other hand, combines the "push" mechanism with a "pull." As operands are produced, they are pushed into the CAM, which acts as both main memory and buffer space. When the control system schedules a new Kernel for execution, requests are issued to the CAM for certain tags or tag ranges in a pull fashion, which eliminates the need for extra buffering of unmatched operands since only

immediately useful operands are requested. This is more analogous to the vector machine technique of predicting memory accesses through pipelined memory accesses in order to hide latency. The result gives data dependent memory access patterns with high throughput. That difference will be assumed as other architectures are discussed below.

Hong and Payne described a hybrid dataflow model that structurally resembles Greedy CAM [22]. That model also partitions the dataflow graph into larger kernels, but treats the tags themselves similarly to previous dataflow machines, in that wildcard and functional tag relationships are not supported as they are in Greedy CAM. Further, the conditional execution and function call mechanisms appear to be more traditional in nature, as compared to Greedy CAM's.

Another hybrid dataflow machine is the TRIPS architecture, which also splits a dataflow program into sub graphs and dynamically schedules these at runtime [23]. If CAM and Operand Set hardware components were added to the TRIPS system, it would make a good Greedy CAM engine, particularly in the S-Morph configuration. The addition of the Greedy CAM's abstraction of memory would help alleviate any dependency on a low latency memory system. There are also differences in the scheduling mechanism, which appears to be control driven in TRIPS. Greedy CAM allows a more distributed control system through the use of Signals that can activate Kernels in more flexible orders.

The WaveScalar architecture is another interesting dataflow computer [24]. In a sense, it also breaks the dataflow graph into sub-graphs, the elements of which are distributed across many execution units on the chip, to be fetched from small caches as needed. The execution units and caches are networked to pass results back and forth as needed. Unlike Greedy CAM, all the inputs to the sub-graph are not required to enter it in lock step, which can result in some paths executing faster than others, hopefully generating some memory requests before the slower paths block on those responses. The downside is that the faster paths can run too far ahead of the slower ones causing a buffering problem, which the authors cite as being the subject of future work. In the event that the input buffers overflow, the values are spilled into external memory and then retrieved later when a possible match arrives on another input. Greedy CAM solves the buffering problem by assembling Operand Sets as a unit. As each Operand Set is built, it is released into the execution hardware without having to consider the

possibility of stalls or buffering due to missing operands. Individual operations, or instructions, in WaveScalar are called up from the caches as needed, possibly from main memory in the event of a miss, in response to demand for those instructions as targets. Hence, WaveScalar is a finer grained, more dynamic, system than Greedy CAM, though the authors note that the path to scaling for large parallelism is not clear. They suggest the possible introduction of threads to address this issue. Greedy CAM works in coarser grained parallelism, which leads to less communication overhead between operations. Rather than needing to synchronize through caches, Greedy CAM operators are typically directly connected through predefined paths. Further, Greedy CAM allows the parallel execution of several Kernels simultaneously with program specified guarantees of data independence. The Greedy CAM environment should lend itself more readily to scaling.

Based loosely on STARAN [25], Jerry Potter, et. al., at Kent State University has proposed the ASC machine based on associative computing [26,27,28]. ASC can be thought of as an SIMD machine with associative memory extensions. Both Greedy CAM and the ASC architecture are similar in that an associative memory is used for data storage. However, ASC contains a one bit processing element for each memory word location, and the search operation occurs by all of these processing elements working in SIMD fashion. Each word also possesses a number of flags to include or exclude items from the current data set, making the memory more intelligent than what is being proposed by Greedy CAM. At the same time, one might imagine that the traffic to/from memory may be reduced in that way. This advantage of ASC should not be ignored, but it is a goal to construct a Greedy CAM computer using commercial RAMs, which would be more difficult using the ASC model. Further, Greedy CAM supports the execution of multiple parallel Kernels in an order that is driven by data availability rather than by a controlling program.

Possibly the closest reconfigurable computer is PipeRench; which virtualizes kernels in a pipelined fashion [29]. That architecture places some of the burden of virtualizing the dataflow graph onto the hardware, which maintains state between kernel swaps to maintain the pipelined flow. Greedy CAM takes an approach that instead places the full burden of partitioning onto the compiler, and uses the parallel execution of kernels on replicated execution hardware to make use of hardware configurations of various sizes. These decisions reduce the state that Greedy CAM needs to manage within the execution hardware. Further, PipeRench also attempts to reconfigure

each processing element in a single cycle in order to hide the "startup" costs of reconfigurable computing. Greedy CAM overcomes the startup cost of hardware configuration using background configuration while processing large datasets, with the acknowledgement that large datasets are not pervasive across all types of problems, and therefore PipeRench may have an advantage for some programs.

# XI. Conclusion/Future Work

The single greatest barrier to performance today is memory latency. Caching can effectively combat this for small amounts of data, but large datasets thrash the cache into uselessness. Control driven machines can attempt to overcome this latency by running more threads to generate more parallel accesses and then quickly context switch between them as results return. This requires that enough threads exist to generate memory requests, and that the hardware can maintain enough internal state for each thread for fast context switches. Dataflow machines run many operations in parallel, but generally ignore memory latency, instead relying on many parallel pending instructions to "fire" to keep the memory pipeline full. The Greedy CAM architecture makes an explicit effort to overcome memory latency by building the hardware computational model around the data fetch operation rather than around the instruction fetch operation. This makes it unique among other computational paradigms.

Said another way, there are two processes in any given computational problem. First, the data items in the computation must be brought physically proximate to one another in the hardware. Second, the required arithmetic operation must be performed. The most common architectures of today blend the operations seamlessly into the same instruction stream, running through the same ALU. The Greedy CAM architecture explicitly pulls these two operations apart in an effort to enable greater parallelism and tolerance of latency. Hopefully at the same time, more intuitive programming methodologies become possible where the programmer can merely specify how data elements in the program should go together rather than single stepping through every single one of them.

To date, exploratory simulators have been created in Verilog and C#. These have been used to implement matrix multiply, heap sort, bin sort, Prim's algorithm, and Viterbi decoding. They have helped highlight where improvements can be made in the details of the architecture. A hardware prototype using FPGAs is underway.

The first stage of hardware design will be the design of a CAM unit that can support the features described above. After that the Kernel Execution Unit and Control units will follow. In tandem with this effort, a compiler is being written to target the architecture. These efforts should result in an interesting platform for Greedy CAM experimentation.

It may also be interesting to write a Greedy CAM execution engine that runs on a multicore processor. In that scenario, multiple threads running on different processors could each execute the code for a given Kernel. There would be multiple ways of handling the Operand Set assembly process for each Kernel as well as the creation of the output tags and their associated results. Thus, Greedy CAM may become more than a particular hardware instantiation, but in fact the execution paradigm can be applied to many different architectural styles.

Further information concerning programming one instantiation of the Greedy CAM architecture may be found in a Microsoft Technical Report by Erik Ruf [30].

## Acknowledgements

I would like to thank and acknowledge the contributions and support of Jim Kajiya, Erik Ruf, Mike Sinclair, and Turner Whitted in the development of this architecture. They have been a consistent sounding board for my work and have offered ideas for improvement and a critical view to drive me towards better solutions.

---

[1] S. Vajapeyam, M. Valero. Early 21st Century Processors. In *IEEE Computer Magazine*, pages 47-50, Volume 34, Number 4, April 2001.

[2] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, D. Burger. Clock rate versus IPC: the end of the road for conventional microarchitectures. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, Volume 28, Issue 2, May 2000.

[3] Y. Taur. CMOS Design Near The Limit Of Scaling. In *IBM Journal of Research and Development*, Pages 213-222, Volume 46, Number 2/3, March/May 2002.

[4] K. Hwang. *Advanced Computer Architecture: Parallelism, Scalability, Programmability*, McGraw-Hill, Pages 310-315.

[5] M. V. Wilkes. Slave Memories and Dynamic Storage Allocation. In *IEEE Transactions on Electronic Computers*, Page 270-271, Volume EC-14, April 1965.

[6] R. Tomasulo. An Efficient Algorithm For Exploiting Multiple Arithmetic Units. In *IBM Journal of Rsearch & Development*, Pages 25-33, Volume 11, 1967.

[7] J. E. Smith. Dynamic Instruction Scheduling and the Astronautics ZS-1. In *IEEE Computer Magazine*, Pages 21-35, Volume 22, Issue 7, July 1989.

[8] J. Smith. A Study of Branch Prediction Strategies. In *8th International Symposium on Computer Architecture*, 1982.

[9] J. K. F. Lee, A. J. Smith. Branch Prediction Strategies and Branch Target Buffer Design. In *IEEE Computer Magazine*, pages 6-22, January 1984.

[10] M. Johnson. *Superscalar Microprocessor Design*. Prentice Hall, Englewood Cliffs, NJ, 1990.

[11] B. Calder, G. Reinman. A Comparative Survey of Load Speculation Architectures. In *Journal of Instruction-Level Parallelism*, Volume 2, May 2000.

[12] W. Zhang, S. Checkoway, B. Calder, D. M. Tullsen. Dynamic Code Value Specializatino Using the Trace Cache Fill Unit. In *24th International Conference on Computer Design*, October 2006.
[13] N. Tuck, D. M. Tullsen. Multithreaded Value Prediction. In *11th International Symposium on High Performance Computer Architecture*, February 2005.
[14] Micron MT47H32M16CC-37E DDR2 RAM Datasheet. http://www.micron.com
[15] R. Hartenstein. Coarse Grain Reconfigurable Architectures. In *Proceedings of the 2001 Conference on Asia South Pacific Design Automation*, pages 564-570, 2001.
[16] S. Ahuja, N. Carriero, D. Gelernter. Linda and Friends. In *IEEE Computer*, Pages 26-34, Volume 19, Issue 8, August 1986.
[17] V. Krishnaswamy, S. Ahuja, N. Carriero, D. Gelernter. The Architecture Of A Linda Coprocessor. In *Proceedings of the 15th Annual International Sumposium on Computer Architecture*, Volume 16, Issue 2, May 1988.
[18] A. H. Veen. Dataflow Machine Architecture. In *ACM Computing Surveys*, Pages 365-396, Volume 18, Number 4, December 1986.
[19] D. E. Culler. TR-332 Resource Management for the Tagged Token Dataflow Architecture. MIT Technical Report available from http://libraries.mit.edu/guides/types/techreports/#online
[20] J. R. Gurd, C. C. Kirkahm, I. Watson. The Manchester Prototype Dataflow Computer. In *Communications of the ACM*, Volume 28, Number 1, January 1985.
[21] G. M. Papadopoulos, D. E. Culler. Monsoon: An Explicit Token-Store Architecture. In Proceedings of the 17th Annual International Symposium on Computer Architecture, Volume 18, Issue 3a, 1990.
[22] Y. C. Hong, T. H. Payne. A Hybrid Approach for Efficient Dataflow Computing. In *Ninth Annual International Phoenix Conference on Computers and Communications*, Pages 170-178, March 1990.
[23] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, N. Ranganathan, D. Burger, S. W. Keckler, R. G. McDonald, C. R. Moore. TRIPS: A Polymorphous Architecture for Exploiting ILP, TLP and DLP. In *ACM Transactions on Architecture and Code Optimization*, Pages 62-93, Volume 1, Number 1, March 2004.
[24] S. Swanson, K. Michelson, A. Schwerin, M. Oskin. WaveScalar. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, December 2003.
[25] K. E. Batcher. Flexible Parallel Processing and STARAN. In *WESCON Technical Papers*, Pages 1/5-1 - 1/5-3, September 1972.
[26] J. L. Potter. *Associative Computing, A Programming Paradigm for Massively Parallel Computers*. Plenum Press, New York and London, 1992.
[27] J. Potter, J. Baker, S. Scott, A. Bansal. ASC: An Associative Computing Paradigm. In *IEEE Computer Magazine*, pages 19-25, Volume 27, Issue 11, November 1994.
[28] R. Walker, J. Potter, Y. Wang, M. Wu. Implementing Associative Processing: Rethinking Earlier Architectural Decisions. In *Proceedings 15th International Parallel and Distributed Processing Symposium*, pages 2092-2100, April 2001.
[29] S. C. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. R. Taylor, R. Laufer. PipeRench: A Coprocessor for Streaming Multimedia Acceleration. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, Volume 27, Issue 2, May 1999.
[30] E. Ruf. *Programming the Greedy CAM Machine*. Microsoft Technical Report #MSR-TR-2007-04, http://research.microsoft.com/research/pubs/ or ftp://ftp.research.microsoft.com/pub/tr/TR-2007-04.pdf.