Programming Multi-core Architectures Using Data-Flow Techniques[1]

Samer Arandi
Department of Computer Science
University of Cyprus
Nicosia, Cyprus
Email: samer@cs.ucy.ac.cy

Paraskevas Evripidou
Department of Computer Science
University of Cyprus
Nicosia, Cyprus
Email: skevos@cs.ucy.ac.cy

***Abstract*—In this paper we present a Multithreaded programming methodology for multi-core systems that utilizes Data-Flow concurrency. The programmer augments the program with macros that define threads and their data dependencies. The macros are expanded into calls to the run-time that creates and maintains the dependency graph of the threads and performs the scheduling of the threads using Data-Flow principles.**

We demonstrate the programming methodology and discuss some of the issues and optimizations affecting the performance. A detailed evaluation is presented using two applications as case studies. The evaluation shows that the two applications scale well and compare favorably with the results of similar systems. Our results demonstrate that Data-Flow concurrency can be efficiently implemented as a Virtual Machine on multi-core systems.

I. Introduction

Over the last five decades high-performance was achieved by relying on improvements in fabrication technologies and architectural/organizational optimizations. However, the most severe limitation of the sequential model, namely its inability to tolerate long latencies, has slowed down the performance gains, forcing the industry to hit the Memory wall. As a result of this and other factors, such as the Power Wall and the diminishing returns of Instruction Level Parallelism, the entire industry had to switch to multiple cores per chip. This ushered the beginning of the Concurrency Era, as it soon became evident that traditional programming models did not allow for efficient utilization of the large number of resources now available on a single chip.

The Data-Flow model [1], [2], [3] is an alternative model that handles concurrency and tolerates memory and synchronization latencies efficiently. Moreover, the side-effect free semantics of Data-Flow exposes the maximum potential parallelism in programs by enforcing the minimum amount of ordering on execution (i.e. only enforce true data dependencies).

In this work we present a programming methodology based on the Data-Driven Multithreading (DDM) [4], [5], [6] model of execution that combines Dynamic Data-Flow concurrency with efficient sequential execution. The programming methodology targets the Data-Driven Multithreading Virtual Machine (DDM-VM).

C Programs are augmented with a set of *macros* that define:

- Thread boundaries
- Producer-consumer relationships amongst the threads
- The data produced and consumed by each thread

The macros expand to calls to the runtime of the DDM-VM to manage the execution according to the DDM model. The macros represent the low-level programming abstraction of the DDM-VM. While currently the programmer adds the macros to the code by hand, two compiler projects, currently under development will automate this task. The first is based on Concurrent Collections (CnC) [7] a platform-independent, high-level parallel language, with the help of a source-to-source compiler. The second is a GCC-based auto-parallelizing compiler for the C Language.

This paper presents a macro-based approach for programming multi-core architectures using Data-Flow techniques. The approach is demonstrated in detail and some of the factors and optimizations affecting the performance are discussed. An in-depth evaluation of two case study applications highlights the effect of the factors on performance and shows that both applications scale well and outperform the results of similar systems.

II. Data-Driven Multithreading Virtual Machine (DDM-VM)

Data-Driven Multithreading (DDM) is an execution model that combines the benefits of the Data-Flow model in exploiting concurrency with the efficient execution of the control-flow model. DDM decouples the execution from the synchronization part of a program and allows them to execute asynchronously, thus tolerating synchronization and communication latencies efficiently.

The core of the DDM model is the Thread Scheduling Unit (TSU) [8] responsible for scheduling threads at run-time based on data-availability. DDM utilizes data-driven caching policies [9] to implement deterministic data prefetching which improves locality.

The Data-Driven Multithreading Virtual Machine (DDM-VM) is a virtual machine that supports DDM execution on

[1] This work was supported by the Cyprus Research Promotion Foundation under Grants ΔΠ/0505/25E & ΠΕΝΕΚ/ΕΝΙΣΧ/0308/44

homogeneous and heterogeneous multi-core systems. DDM-VM virtualizes the parallel resources of the underlying machine and uses a general, unified representation for DDM programs. The representation is flexible enough to incorporate additional information needed to optimize the DDM execution for different target architectures. The DDM-VM runtime system, composed mainly of the TSU, handles the tasks of thread scheduling, execution instantiation and data prefetching implicitly.

DDM-VM programs consist of two parts: the code of the DDM threads and the *synchronization graph* describing the consumer-producer dependencies amongst the threads. At the start of a DDM-VM application execution, the runtime is spawned on the cores and the execution of the TSU is started. The dependency graph is loaded into the TSU and used to schedule threads based on data-availability. When a thread finishes execution the TSU uses the graph to identify and update its consumers. Once the data of a consumer thread is produced, the CacheFlow module in the TSU prefetches that data from main memory to the cache of the processor where the consumer thread is scheduled to run. After the prefetching completes the thread is ready to be started.

The DDM-VM architecture currently has two implementations. The Data-Driven Multithreading Virtual Machine for the Cell [10] (DDM-VM$_c$) and the Data-Driven Multithreading Virtual Machine for Symmetric Multi-cores (DDM-VM$_s$). Figure 1 depicts both implementations.

The DDM-VM$_c$ is the DDM-VM implementation targeting heterogeneous multi-cores with software-managed memory and has the Cell as the main target. The Cell processor is a heterogeneous multi-core chip composed of one general-purpose RISC processor called the Power Processor Element (PPE) and eight fully-functional SIMD co-processors called the Synergistic Processor Elements (SPE). Each SPE has its own 256KB software-controlled local store (LS) memory and can only execute instructions and access data present in the LS .

In DDM-VM$_c$, the TSU is implemented as a software module running on the PPE core, while the execution of the threads takes place on the SPE cores. The structures holding the synchronization information and the state of the TSU are allocated in main memory including part of the structures required for the operation of the Software CacheFlow (S-CacheFlow). The code of the DDM threads linked with the runtime library is located in the LS memory in addition to the other S-CacheFlow structures. Part of the LS is pre-allocated and divided into blocks to hold the data of the threads. We refer to this part of the LS as the *DDM Cache*.

The part of the runtime on the SPEs and the TSU communicate via Direct Memory Access (DMA) calls. The S-CacheFlow module in the TSU manages data transfers between the main memory and the LS transparently to the programmer.

The DDM-VM$_s$ is the DDM-VM implementation targeting homogeneous multi-cores. The TSU is also implemented as a software module running on one of the cores, while the threads execution takes place on the other cores.

The part of the runtime running on the cores communicate with the TSU by changing the state of the TSU structures allocated in the shared main memory.

III. DDM-VM Programming Methodology

The DDM-VM utilizes the distributed synchronization mechanism of dynamic Data-Flow as described by the U-Interpreter [2]. Dynamic Data-Flow requires that each data value is associated with a unique tag; a *Token* V[c,s,i] is made up of the value V and the tag [c,s,i], "c" is the context identifier, "s" is the destination address and "i" is the iteration identifier. The U-interpreter provides a formal distributed mechanism for the generation and management of the tags at execution time.

We briefly explain the basic principles of the U-Interpreter with the aid of the inner product example shown in Figure 2-a, the corresponding dynamic Data-Flow graph is shown in Figure 2-b. The equivalent DDM graph is shown in Figure 3 and is made of three threads T1, T2 and T3. The [L] operator in Figure 2-b creates a new loop context by adding a loop identifier to the *tag* and initializing it to zero. Every time a token goes around in the loop the [D] operator increments the iteration part of the *tag*. T1 part of Figure 2-a demonstrates the process of *tag* creation for the first iteration of the loop. Note that in dynamic Data-Flow an actor can fire only when inputs with identical *tags* are available at all its input ports. The whole process is repeated until the loop predicate evaluates to false, the switch actor then routes the last token to the [D^{-1}] operator which restores the iteration part of the *tag* to zero and then the [L^{-1}] operator remove the loop identifier added by the [L] operator thus restoring the *tag* it had before entering the loop.

In DDM, the tagged token matching operations are implemented using synchronization points in virtual memory. The tag manipulation operators are implemented by the runtime.

A DDM program is represented as a number of re-entrant, inter-dependent DDM threads, along with each thread's *meta-data* (or *synchronization template* in DDM terminology). The threads are identified by the *ThreadID* and *Context*. The *context* corresponds to the tag of the U-Interpreter and uniquely identifies dynamic instantiations of each thread. When mapping loops into DDM threads we derive the value of the *context* from the nested loop indices.

The synchronization template of the thread specifies the following attributes:

- **The Instruction Frame Pointer (IPF):** points to the address of the first instruction of the thread.
- **The ReadyCount (RC):** a value equal to the number of producer-threads this thread needs to wait for until starting to execute.
- **The Data Frame Pointer List (DFPL):** a list of pointers to the data inputs assigned for the thread.
- **The Consumer List (CL):** a list of this thread's consumers that is used to determine which *ReadyCount* values to decrement after the thread completes its execution.

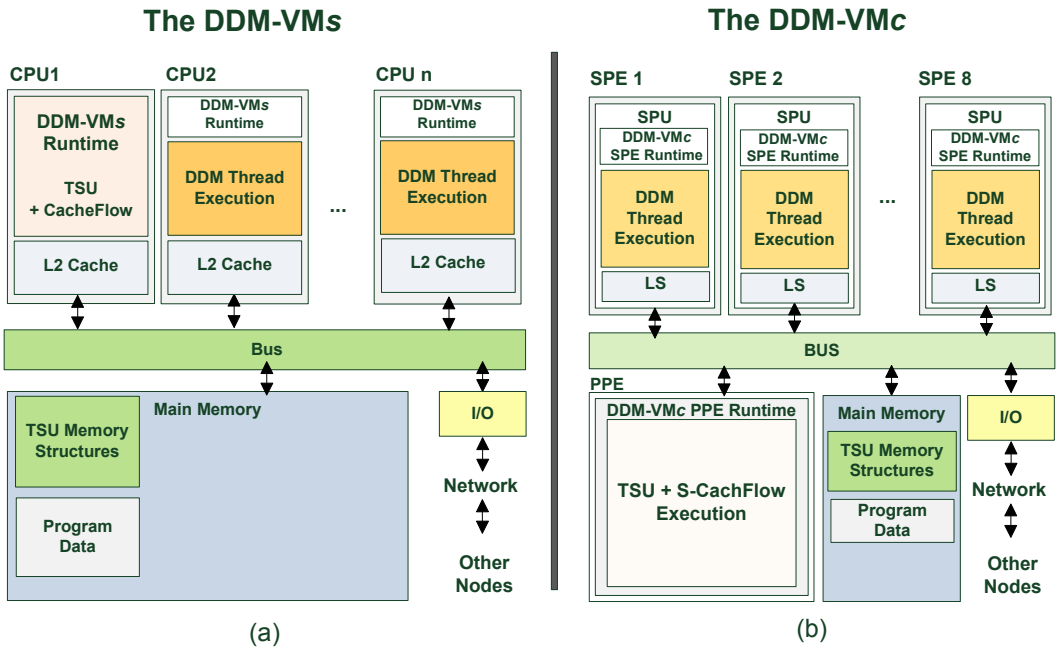

Fig. 1. The current implementations of the DDM-VM architecture (a) DDM-VM$_s$ (b) DDM-VM$_c$

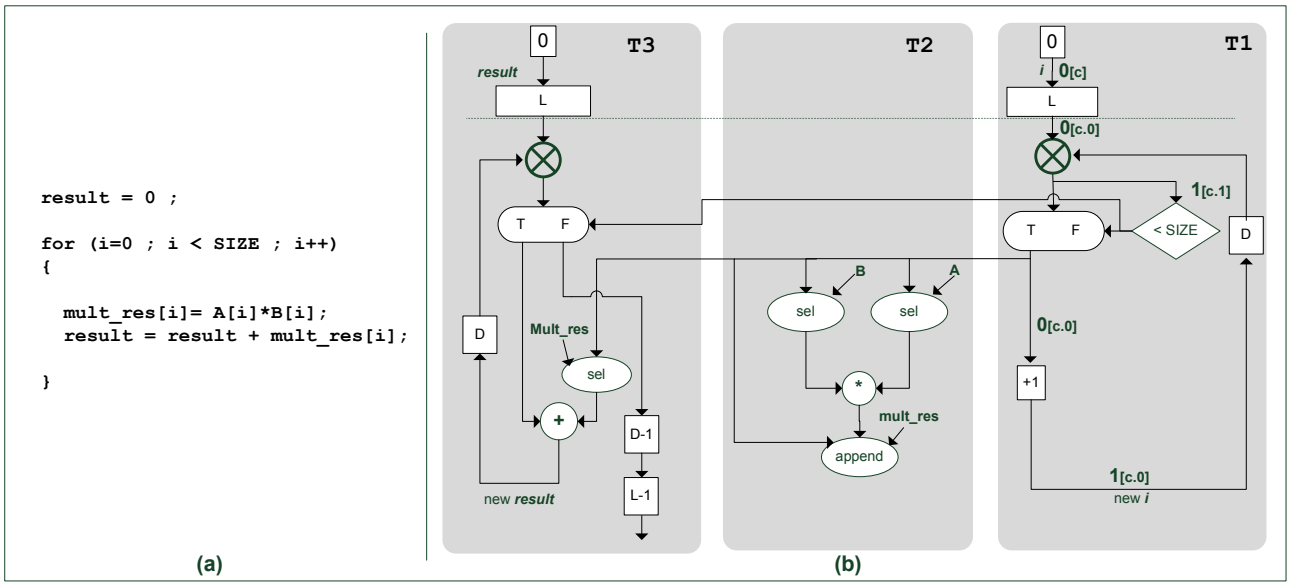

Fig. 2. The Vector Dot Product (a) Original Program (b) U-Interpreter Dynamic Data-Flow Graph

The synchronization templates of all the threads in the DDM program constitute the *data-driven synchronization graph*.

Figure 3 depicts the equivalent DDM graph for the vector dot product program. The outer loop is implemented by DDM thread T1 and the two operations in the body of the loop are implemented by threads T2 and T3, respectively. Figure 3-a illustrates a detailed view of the graph. Solid arrows represent data dependencies and actual movement of data instances between the threads. Dotted arrows represent only data dependencies amongst the threads. Shaded rectangles represent operations, non-shaded single rectangles represent static values and non-shaded multiple rectangles represent dynamic values.

Thread T1 generates the values of the loop index *i*. The shaded portion of T1 generates the *context* value and corresponds to the [D] operator of the U-interpreter of Figure 2-b. The reader will notice that in this example the *context* value is identical to the loop index value. In such cases we omit the extra graph that generates the *context* and use the loop index for both. However, in the general case we need to generate the *context* values that correspond to each invocation of the loop. Same applies for recursive constructs.

In DDM the TSU uses the *context* value to identify the

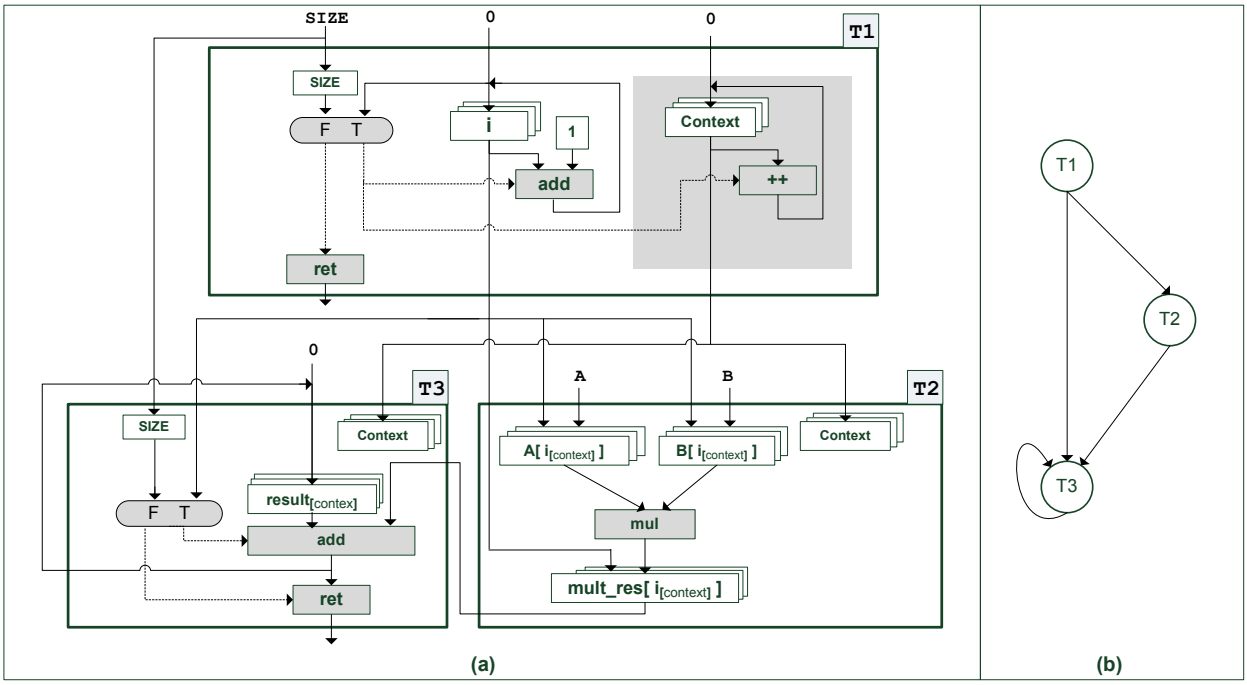

Fig. 3. The Vector Dot Product DDM Dependency Graph (a) Detailed view (b) High-level view

corresponding *readycount* of the invocations of threads T2 and T3 and decrement it. The *context* is also used during the execution of a DDM thread in order to point to the corresponding data value(s).

Thread T2 performs the multiplication and uses the value of *i* to select the appropriate elements of the input and output vectors. T2 then notifies the TSU to update the *readycount* of the invocation of thread T3 consuming the produced value. Thread T3 accumulates the multiplication of the two vectors in *result* and updates (through the TSU) the readycount of the next invocation of T3. When the loop predicate evaluates to false *result* contains the vector dot product value.

DDM-VM represents DDM programs using a set of C *macros* that expand into calls to the DDM-VM runtime. The macros are described in Table I. The process of mapping a program using the DDM-VM macros is explained with the aid of the blocked matrix multiplication on the DDM-VM$_c$ implementation. The code of the original program is depicted in Figure 4-a. We chose the width of matrix B to be equal to one block for simplification.

A. Identifying the Boundaries of DDM Threads

Figure 4-b depicts one possible mapping of this application to a DDM program. The outer *for* loop is mapped into a DDM thread called *THREAD_1* and the inner loop and its body are mapped into a DDM thread called *THREAD_2*. The limited space of the LS precludes us from executing the entire row times the column. Thus, we execute the multiplications in blocks. Each instance of *THREAD_2* calls the *MultAdd_Block* kernel that performs the operation *C[i]=C[i]+A[i][j]*B[j]*. This multiplication-accumulation introduces a dependency between the successive iterations of the inner loop. Thus, no parallelism can be exploited by unraveling this loop. For that reason the index generation of the inner loop and its body are merged into a single DDM thread (*THREAD_2*). This is an optimization that avoids the overhead of having a thread that generates the loop indices for a sequential loop.

Figure 4-b illustrates the dynamic instantiations of the DDM threads with their dependencies represented as arrows. Each instantiation is labeled with its *context* as <*context*>.

The corresponding code of the DDM-VM$_c$ program threads is shown in Figure 5. The macros *DVM_THREAD_START* and *DVM_THREAD_END* mark the boundary of the threads. The *DVM_LOOKUP(TYPE,VAR)* macro retrieves the LS address of one data input/output of the thread and stores it in the pointer variable *VAR* after type-casting it to *TYPE*. The *DVM_CONTEXT* is a variable set by the runtime to the current value of the *context*. The *GET_CONTEXT* macro is used to retrieve the *context* components corresponding to the values of the two loop indices *i* and *j*.

B. DDM Dependency Graph and Context Maintenance

The *DVM_UPDATE* macro informs the TSU to decrement the *readycount* of the thread consumers. It specifies the dynamic instantiation of the consumer to update by designating the *context* value of that instantiation using the *MAKE_CONTEXT* macro in combination with a number of operators. The last two macros are used to implement the U-Interpreter *context* operators.

The for loop in the body of *THREAD_1* implements the [D] operator by incrementing the loop index *i*. For every value of *i*, the thread invokes the DVM_UPDATE macro

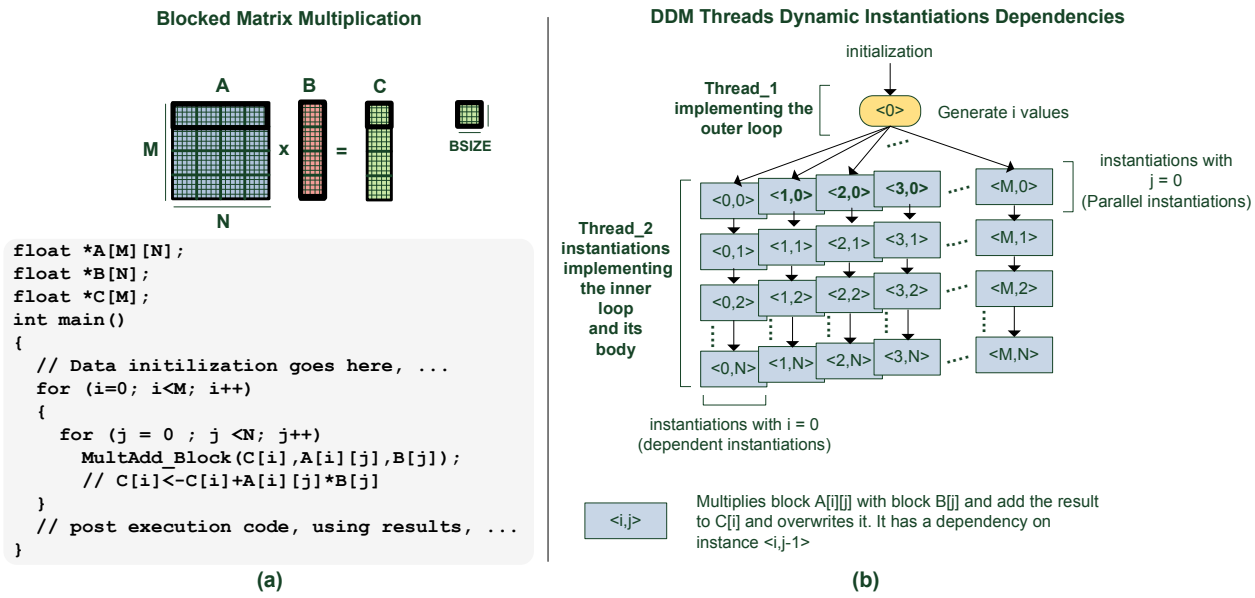

Fig. 4. The Blocked Matrix Multiplication Application (a) The original code of the application (b) Dependencies across the dynamic instantiations of the DDM threads

```
DVM_THREAD_START(TID_THREAD_1);

for (i = 0 ; i < M ; i++)
  DVM_UPDATE(CONS1,OP_SET_CONTEXT,MAKE_CONTEXT_D(i,0));

DVM_THREAD_END();
```

```
DVM_THREAD_START(TID_THREAD_2);
DVM_LOOKUP(float *,A);
DVM_LOOKUP(float *,B);
DVM_LOOKUP(float *,C);
GET_CONTEXT_D(DVM_CONTEXT,i,j);

MultAdd_Block(C,A,B);

if (j < N-1)
   DVM_UPDATE(CONS1,OP_INC_INDX,0);//update consumer
DVM_THREAD_END();
```

Fig. 5. The code for the DDM threads using the DDM-VM macros

and passes the OP_SET_CONTEXT operator as a parameter which implements the U-Interpreter [L] operator. This operator creates the *context* of the instantiations of *THREAD_2* by informing the TSU to update the *readycount* of the first instantiation of every group of the dependent instantiations of *THREAD_2*.

C. DDM Dependency Graph Creation and Execution

Figure 6 depicts the code that runs before and after the execution of the DDM threads in the program. After data initialization, the programmer uses the *DVM_CREATE_THREAD_TEMPLATE* macro to load the *synchronization template* of each DDM thread into the TSU.

The programmer then uses the *DVM_UPDATE_THREAD* macro (which is used to decrement the *readycount* of a specific thread directly) to decrement the readycount of *THREAD_1* making it ready for execution, before calling the

```
float *A[M][N];
float *B[N];
float *C[M];
int main()
{
  // Data initilization goes here
  // THREAD_1 DDM thread template
  DVM_CREATE_THREAD_TEMPLATE(TID_THREAD_1,     //TID/IFP
                             TID_THREAD_2,0,    //consumers
                             1,                 //readycount=1
                             0,                 //DFPNum=0
                             DVM_DYNAMIC,0,     //scheduling Mode
                             DVM_ARITY_0,       //nesting-level=0
                             DVM_ASSOCIATIVE,0);//Associative SM

  // THREAD_2 DDM thread template
  DVM_CREATE_THREAD_TEMPLATE(TID_THREAD_2,     //TID/IFP
                             TID_THREAD_2,0,    //consumers
                             1,                 //readycount = 1
                             3,                 //DFPNum = 3
                             DVM_CUSTOM,0,      //Scheduling Mode
                             DVM_ARITY_2,       //nesting-level=2
                             DVM_ASSOCIATIVE,0);//Associative SM

   DVM_UPDATE_THREAD(TID_THREAD_1,0); // update the
                  // readycount of thread THREAD_1 (becomes zero)
    DVM_EXECUTE(); // start the execution of the TSU, only
                   //comes back after all threads finish
}
```

Fig. 6. Initialization, graph creation, graph execution and post-execution code

DVM_EXECUTE macro which starts the execution of the TSU and the scheduling of threads to the cores.

a) The Data of the Threads: One of the parameters of the the *DVM_CREATE_THREAD_TEMPLATE* macro sets the number of entries in the Data Frame Pointer List (DFPL) of the thread. However, the information of the DFPL itself is encoded by another set of macros shown in Figure 7. The macros specify the address, size and flags for the data of the thread. The information of the data is directly extracted from the original code. The only difference is that the loop indices

TABLE I
THE DDM-VM MACROS

DDM-VM macros	Operation
DDM Thread Boundaries	
`DVM_THREAD_START(THREAD_ID)`	identifies the first instruction of the thread and assigns the thread identifier
`DVM_THREAD_END()`	• informs the TSU that the thread has finished execution • relinquishes control to the runtime to execute the next ready thread
`DVM_LOOKUP(TYPE,VAR_NAME)`	• perform DDM Cache Lookup to get the address of the thread data in the LS
DDM Dependency Graph Creation	
`DVM_CREATE_THREAD_TEMPLATE(THREAD_ID, CONS1,CONS2, READY_COUNT, NUMBER_OF_DFPs, SCHED_MODE,SCHED_VALUE, ARITY, SM_METHOD,SM_VALUE)`	creates and loads the thread synchronization template consisting of: the thread identifier, the consumers (if the thread has more than two consumers CONS1 = 0 and CONS2 is a pointer to a list of consumers), the *readycount* value, the number of DFPs, the scheduling policy for the thread, the *arity* which indicates the loop nesting level of the thread and the Synchronization Memory (SM) method to use.

`DVM_START_DFPL(THREAD_ID)`
`DVM_SET_DFP(ADDR,SIZE,FLAG)`
`DVM_END_DFPL()`
	assigns the information of the DFPL: the address and size of the input/output data of the thread. The flag field specifies the direction of data access (in, out or inout) and the re-use flags when exploiting locality.

`DVM_SET_REFCOUNT(VALUE1,VALUE2)`	assigns the reference-counter values for data items that will be re-used when exploiting locality
DDM Dependency Graph Maintenance and Context Management	
`DVM_UPDATE(CONS,OP,VALUE)`	• informs the TSU which specific invocation of a consumer thread to decrement its *readycount* when the thread finishes execution • implements the U-Interpreter *context* manipulation operators • any produced data is exported to main memory before executing the update
`DVM_UPDATE_MULTIPLE(CONS,VALUE1,VALUE2)`	• informs the TSU to decrement the *readycount* of multiple invocations of a consumer thread.
`DVM_UPDATE_THREAD(THREAD_ID,VALUE)`	informs the TSU to decrement the *readycount* of a specific invocation of a thread. This macro is used to update threads that consume initialized data
`DVM_CONTEXT`	a variable set by the runtime to give the programmer access to the value of the thread *context*.
`MAKE_CONTEXT(…)/GET_CONTEXT(…)`	create/retrieve the value of the *context*. The value could have single, double, or triple arity.
Thread Scheduling Policy	

`DVM_START_SCHEDULE(THREAD_ID)`
`DVM_SET_SCHEDULE(CoreID)`
`DVM_END_SCHEDULE()`
	overrides the default scheduling policy and controls to which core the invocations of the threads are scheduled

Execution	
`DVM_EXECUTE()`	Starts the execution of the TSU and the scheduling of threads to execution units. It returns after all the DDM threads in the program have finished execution

```
// DFPL definitions
DVM_CACHEFLOW_DFPL_START();//start of function called by TSU
int i,j;
DVM_START_DFPL(TID_THREAD_2); // start of DFPL definition
  GET_CONTEXT_D(DVM_CONTEXT,i,j);
  DVM_SET_DFP(A[i][j],BSIZE*BSIZE*4,DATA_IN);
  DVM_SET_DFP(B[j],BSIZE*BSIZE*4,DATA_IN);
  DVM_SET_DFP(C[i],BSIZE*BSIZE*4,DATA_IN|DATA_OUT);
DVM_END_DFPL();         // end of DFPL definition
DVM_CACHEFLOW_DFPL_END(); // end of function called by TSU
```

Fig. 7. DFPL definition macros

```
// Optional Scheduling policy definition
DVM_SCHEDULING_POLICY_START();//start of function called by TSU
int i,j;
DVM_START_SCHEDULE(TID_THREAD_2);
  GET_CONTEXT_D(DVM_CONTEXT,i,j);
  DVM_SET_SCHEDULE(i%NUMBER_OF_CORES);
DVM_END_SCHEDULE();
DVM_SCHEDULING_POLICY_END(); //end of function called by TSU
```

Fig. 8. Scheduling policy definition macros

used to index the data arrays are replaced by the corresponding components of the *context*.

b) Scheduling Policy: Threads are assigned to cores in a manner that maximizes dynamic load-balancing. This default behavior can be selected by passing the value *DVM_DYNAMIC* as a parameter to the *DVM_CREATE_THREAD_TEMPLATE*. If a custom scheduling policy is to be implemented (to take advantage of locality for example) the programmer passes the *DVM_CUSTOM* instead and encodes the policy using a number of macros which specify the ID of the core to which a thread is scheduled. Figure 8 illustrates an example of a scheduling policy that assigns invocations of *THREAD_2* with the same value of *i* to the same core in a modular fashion.

The macros encoding the DFPL and scheduling policy are defined outside the code of the main. The macros expand to helper functions called by the TSU at runtime to retrieve the information the macros encode.

IV. OPTIMIZATIONS FOR DDM-VM EXECUTION

In this section we discuss some of the optimizations deployed to improve the execution of DDM-VM programs and manage the resources.

A. Compound Updates

If we examine the implementation of the code of *THREAD_1* in Figure 5, we find that for each iteration of the loop a call is made to the *DVM_UPDATE* macro to update the *readycount* of one invocation of the consumer thread *THREAD_2*. For every such call a message is sent to the TSU and an entry is inserted in one of the TSU Queues. This pattern where consecutive invocations of a consumer thread are updated occurs frequently in DDM-VM programs. To optimize this operation the special macro *DVM_UPDATE_MULTIPLE* is provided to send a special request to the TSU for decrementing multiple consecutive invocations of a consumer thread. The TSU manages this special request internally in an optimized manner which reduces overheads significantly. Using

this macro doesn't change the semantics of the program nor the synchronization graph. Applying this optimization to *THREAD_1* thread, the code becomes:

```
DVM_THREAD_START(TID_THREAD_1);
  DVM_UPDATE_MULTIPLE(CONS1,MAKE_CONTEXT_D(0,0),
                            MAKE_CONTEXT_D(M-1,0));

DVM_THREAD_END();
```

B. Resource Management

Unlike other techniques that have difficulty extracting parallelism, Dynamic Data-Flow based techniques have the property of exposing the maximum potential parallelism which could overwhelm the resources of the machine. This is a classical problem in Data-Flow execution [11], [12]. In DDM-VM the most critical resources are the TSU and the caches. DDM-VM controls the amount of concurrency to match the available resources both implicitly by the TSU and explicitly at the level of DDM-VM programs.

a) TSU Resource Management: The operations of the TSU matches the status of the TSU resources: TSU queues and structures. When the Firing Queue (FQ) holding the information of ready threads is full or the Local Store (LS) memory of a certain SPE is full, the S-CacheFlow module in the TSU is disabled until some resources are freed. This type of control mechanism is transparent by default, but can be controlled using configuration files for the DDM-VM.

b) Resource Management in Programs: The programmer can apply a technique similar to loop throttling (bounding) [11] to limit the number of thread invocations that are active concurrently. This technique is implemented by introducing auxiliary dependencies in the program graph.

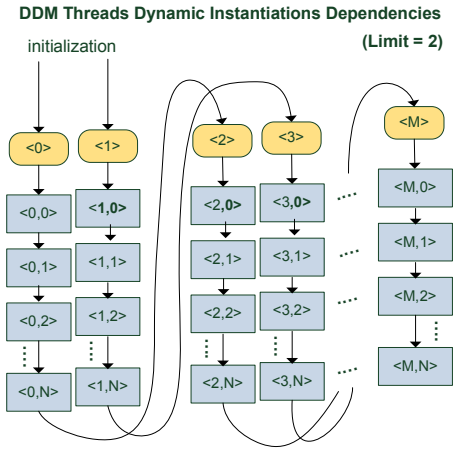

Fig. 9. Dynamic Dependency Graph with throttling - Limit set to 2

Figure 9 illustrates the dynamic dependency graph for the example program after applying this technique on *THREAD_1* implementing the outer loop. The limit value for the throttling is set to 2 in this example. This value can be determined at runtime and adapted to the problem size and number of execution units.

Another technique is to partition the program into DDM Blocks. A DDM block is equivalent to a function or a loop body in the original program, and so, each block contains a subset of the DDM threads in the program. This reduces the demand on the TSU resources as only a subset of the DDM threads will be executing at a given time.

C. Synchronization Memory Optimizations

Dynamic Data-Flow execution involves the generation and consumption of tagged data *tokens* [2] in the system. In DDM all the tagged token matching operations are reduced into virtual memory translations and implemented as updates to the Synchronization Memory (SM) structure allocated in main memory. The SM holds the *readycount* values of the different invocations of the DDM threads.

As the operation of the SM is critical for the performance of DDM execution, we have experimented with 3 different implementations:

- **Direct:** Each invocation of a DDM thread is allocated a unique SM entry. The allocation occurs at the time of creating the thread template. Accessing the entry at runtime is a direct operation that uses part of the *context* to index the SM.
- **Associative:** A standard hashtable is used to allocate the SM entries. The allocation is performed as the execution proceeds. Accessing the entry is an associative operation.
- **Hybrid:** A pre-allocated buffer is used for holding the SM entries. Allocation and de-allocation within the buffer are performed as execution proceeds. Accessing the entry is performed using an associative operation that uses part of the *context* to locate a list of entries in the buffer, followed by a direct operation using the remaining part of the *context* to index the exact entry.

The *direct* implementation involves the least runtime overhead for allocating and accessing the SM entries. However, the programmer is required to provide information on the maximum value the *context* of the thread would reach which is typically conveyed from loop bounds. The *associative* implementation requires no information from the programmer and has no bound on the size it can reach, but its performance depends on the hash function. The *hybrid* implementation takes advantage of locality to reduce the size of the SM by re-using entries.

D. Software CacheFlow Coherency on Software-managed Memory Architectures

Multi-core architectures with software-managed memory hierarchy introduce private address spaces and rely on software to manage data transfers and maintain coherency, which can be both complex and error-prone. The DDM-VM$_c$ manages the memory hierarchy using Software CacheFlow (S-CacheFlow); a fully automated pre-fetching software cache implementing the concept of CacheFlow. A portion of the LS of each SPE is pre-allocated and divided into cache blocks. A Cache Directory (CD) structure is maintained by the TSU per LS to keep track of the blocks state.

S-CacheFlow maintains consistency by prefetching input data from main memory to the LS before the thread starts execution and writing-back produced data to main memory after the thread finishes execution. This scheme reduces expensive CD accesses on the part of the TSU as it expects the data to be *always* in main memory.

To exploit locality, S-CacheFlow can keep the blocks of the data produced by a thread in the LS to be re-used by consumer threads scheduled to run on the same SPE. This is done by passing a special flag (*DATA_KEEP*) to the DDM-VM macro specifying the Data Frame Pointer (DFP) of the produced data. In this case, the data is kept in the LS and the dirty bit of the CD entries of the cache blocks is set to make sure the data will be written back to memory at the end of the program or when the blocks are evicted to allocate space for other threads. Similarly, the consumer threads expected to re-use the data pass a special flag (*DATA_REUSE*) to the macro specifying the corresponding DFP entry. The flag causes S-CacheFlow to perform a lookup on the CD of the SPE where the threads are scheduled to run and if this results in a hit the address in the LS is returned and no fetching of data occurs.

This minimizes overheads typically associated with software caches as only data with the *DATA_REUSE* flag results in lookup operations on the CD. Moreover, since this lookup will be a hit in most cases, the overhead is offset by the benefits of the re-use.

In the cases where it is necessary to specify "when" to write-back data kept for re-use to main memory to guarantee cross-SPE coherence, the following two techniques can be used:

- explicitly inserting "writes" in the graph of the program to make sure data is written-back to main memory before it is required by any consumer running on a different SPE.
- assigning a *reference-count* value for every such data to ensure that the data is written-back to main memory after a specific number of "re-uses" on the current SPE. This technique is conservative; if we cannot determine the value of the *reference-count* at compile time we don't apply it.

V. Performance Evaluation

We demonstrate the effect of the factors discussed on the performance of DDM-VM programs using the Blocked Matrix Multiplication and the Blocked Cholesky factorization as case studies. The results are for the DDM-VM$_c$ implementation running on a Sony Playstation 3 (PS3) machine with Linux 2.6.23-r1 SMP ppc64 OS and the IBM Cell SDK version 2.1. The Cell processor powering the PS3 has 6 SPEs available for the programmer out of the original 8. The block size for the matrices in both applications was 64x64 single precision floating-points.

A. Resource Management

To assess the DDM-VM resource management control mechanisms we have executed two sets of experiments for both benchmarks. In the first, we have varied the size of the Firing Queue (FQ) and in the second, we have utilized Loop Throttling and varied the limit on the number of concurrent invocations of the throttled threads. To neutralize the effect of the FQ on the second set we have chosen a relatively large size for the FQ (FQ=6). Figure 10 depicts the results.

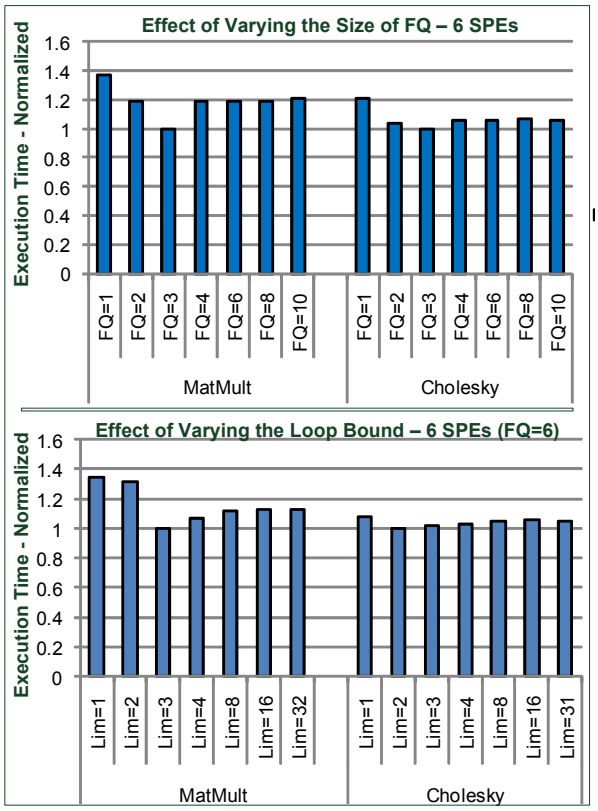

Fig. 10. Resource management control - Effect of Firing Queue (FQ) size and Loop Throttling on performance

In the first set of experiments, the results show that for both applications as the size of the FQ increases the concurrency increases and the performance improves reaching its best when the size is 3. This was expected since the space allocated for the DDM Cache on the LS of each SPE can fit at maximum the data of 3 concurrent invocations of the most computationally intensive threads of the two applications. When the size increases beyond 3 the surplus concurrency causes the performance to degrade. In the second set of experiments utilizing loop throttling, a similar effect to the one in the first set is observed. The effect of throttling on Cholesky is smaller in comparison to MatMult as only one out of the five threads in Cholesky was throttled.

TSU resource control mechanisms like setting the size of the FQ has a global effect that applies to all the threads in the program, while loop throttling can be used to control individual threads for fine tuning of performance.

B. Synchronization Memory (SM) Optimizations

Figure 11 illustrates the performance of the two applications under the 3 different SM implementations.

As expected the *direct* implementation achieves the best performance for both applications as it incurs the minimum

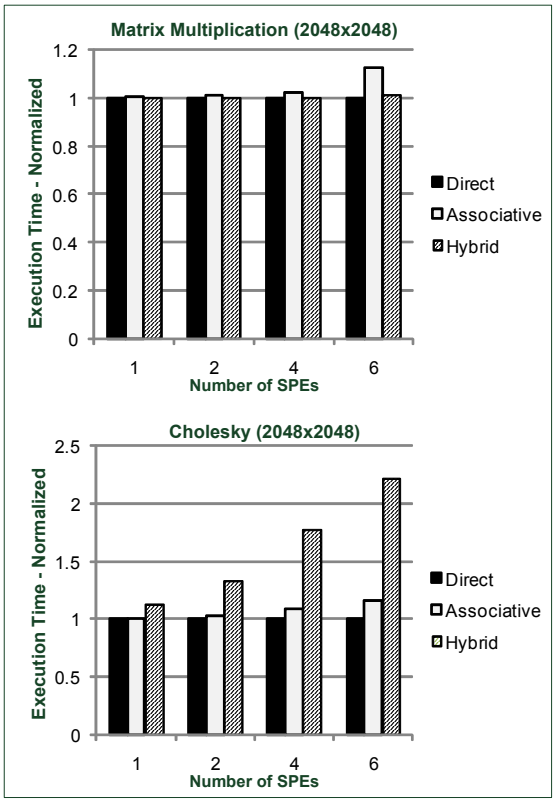

Fig. 11. Effect of the different Synchronization Memory implementations on performance

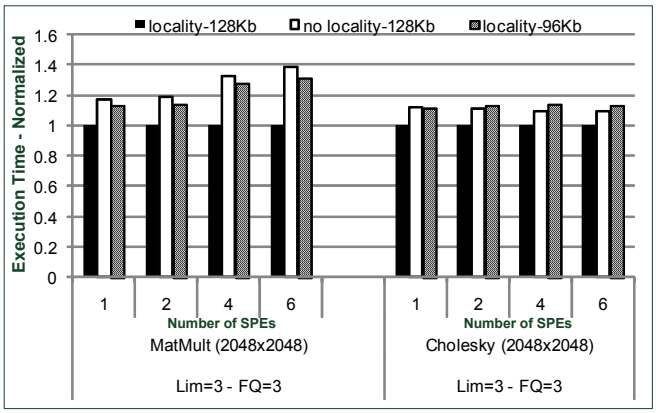

Fig. 12. Effect of locality on performance

overhead for updating the SM entries. The *associative* implementation performs 2nd best on average. The overhead of the associative updates in this implementation increases when the number of cores is high as the TSU is working more in that case. The *hybrid* implementation performs very close to the *direct* and better than the *associative* for MatMult, but performs less than the two other implementations for Cholesky. In MatMult the execution of the threads proceeds consecutively generating regular patterns of updates to the SM which is captured well by the re-use mechanism of *hybrid*. The Cholesky application has a much more irregular pattern of execution which generates non-consecutive updates that cannot be captured well. This results in more allocations and more associative searches that degrade the performance. We are working on a more efficient re-use mechanism for *hybrid* that can capture such patterns. One possible improvement is to utilize information on the expected pattern of the threads execution to guide the re-use of entries.

C. Locality and Data Re-use Exploitation

Figure 12 illustrates the effect of locality. The black and white bars depict the normalized execution time for the two applications with and without locality. The improvement in performance when exploiting locality was achieved by simply identifying the threads that can benefit from locality and adding the *DATA_KEEP* and *DATA_REUSE* flags to their macros and the rest was automated by S-CacheFlow.

It is worthwhile to note that the main source of improvement is the reduced demand of the LS space. Enabling locality for the MatMult, allows the data of 3 invocations of the thread performing the multiplication to fit concurrently in the DDM cache, since one of the input blocks is re-used by all 3 invocations. When locality is not enabled, the data of 2 invocations only can fit. Fitting the data of more threads allows the TSU better chance to prefetch data and overlap latencies with computation which improves the performance. The Cholesky application benefits similarly but to a lesser degree as only one of the computational threads of the application can benefit from re-use.

To confirm our analysis we have executed both applications with locality enabled after reducing the size of the DDM Cache from 128KB to 96KB which has a similar effect on the number of threads that can fit its data concurrently. The results represented by the shaded bar show that the performance degrade in a fashion corresponding to the case when no locality is enabled.

The results demonstrate the deep implications the size of the LS memory has on the execution behavior and consequently the importance of taking into account the size of the working set when choosing the granularity of the threads.

D. Overall Performance

Figure 13 depicts the GFLOPs performance results for the two applications and at the same time we compare our results with that of CellSs [13], a parallel platform targeting the Cell processor. The result for CellSs were obtained by executing the MatMult and Cholesky applications found in the latest release of CellSs platform V2.2 on a PS3. The two applications use the same computational kernels we have used for our applications. For these results we have used the following combination of parameters which produced the best performance. For the MatMult application (FQ=3, Lim=8, Locality enabled, Cache Size=128KB and using the *direct* SM technique). For the Cholesky application (FQ=3, Lim=3, Locality enabled, Cache Size=128KB and using the *direct* SM technique).

The results show that for the MatMult application DDM-VM$_c$ performs very well achieving an average of 88% of the

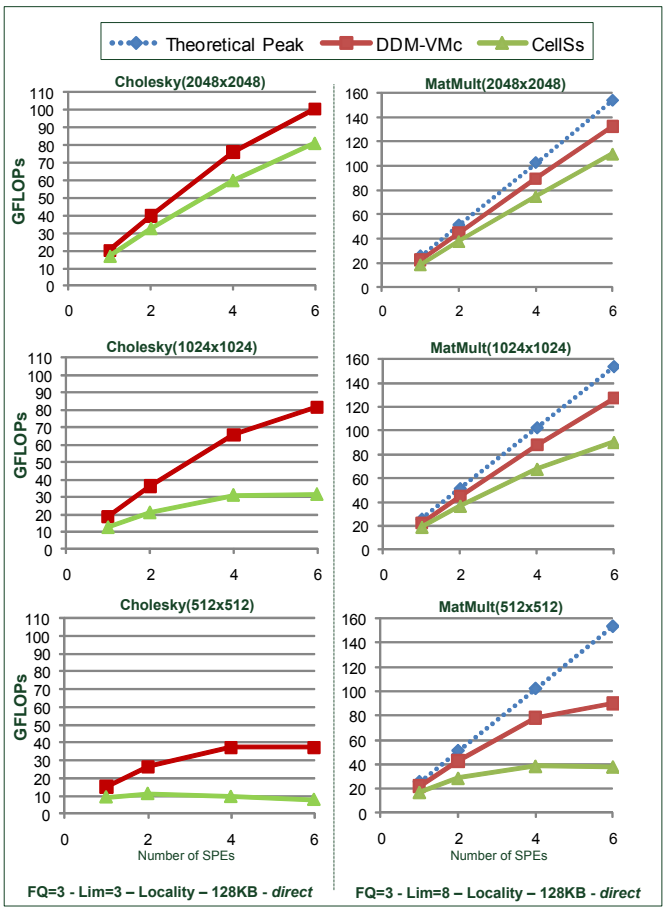

Fig. 13. Performance results

theoretical peak performance for the 2048 size and an average of 86% and 76% for the 1024 and 512 sizes respectively, scaling almost linearly on the first two sizes. The results for Cholesky are not as good as MatMult for the smaller sizes due to the complex nature of the application, however when the size becomes 2048 the application scales very well achieving a speedup of 5 on 6 SPEs.

Comparing the performance with CellSs, DDM-VM$_c$ outperforms it for the entire range for both applications. DDM-VM$_c$ achieves an average improvement of 80% for the 512 size, 28% for 1024 and 19% for the 2048 size for MatMult. An improvement of 213% for 512, 99% for 1024 and 23% for 2048 is achieved for Cholesky. We attribute this to the fact that although CellSs schedules annotated tasks at run-time based on data-dependencies like our model, in contrast with ours which creates the dependency graph statically, CellSs builds it at run-time. This can contribute more delay to the critical path of the application than in our model. Moreover, CellSs makes only part of the graph available to the scheduler and consequently a fraction of the concurrency opportunities in the applications is visible at any time. DDM-VM$_c$ achieves the best improvement v.s. CellSs for the smaller problem sizes, which indicates that it introduces less overhead for exploiting concurrency.

VI. Conclusion and Future Work

In this paper we presented a programming methodology which exploits Data-Flow concurrency in sequential programs with the use of Data-Driven Multithreading macros.

An in-depth analysis of the factors affecting the performance of DDM-VM$_c$ and several optimizations were presented.

Our results demonstrate that Data-Flow concurrency can be efficiently implemented as a VM. The advantage of our model, and our contribution to the state-of-the-art, is that we have implemented Dynamic Data-Flow principles efficiently on off-the-shelf multi-core systems that outperform similar systems.

We are currently working on extending the DDM-VM to support execution across distributed multi-core nodes and developing compilation tools to automate the programming.

Acknowledgment

The authors would like to thank Pedro Trancoso, Costas Kyriacou and Kyriacos Stavrou for their significant contributions in the development of Data-Driven Multithreading on which this work was based.

References

[1] J. B. Dennis, "First Version of a Data Flow Procedure Language," in *Programming Symposium, Proceedings Colloque sur la Programmation*. London, UK: Springer-Verlag, 1974, pp. 362–376.

[2] Arvind and K. P. Gostelow, "The U-Interpreter," *IEEE Computer*, vol. 15, no. 2, pp. 42–49, 1982.

[3] I. Watson and J. Gurd, "A Practical Data Flow Computer," *IEEE Computer*, vol. 15, no. 2, pp. 51–57, 1982.

[4] C. Kyriacou, P. Evripidou, and P. Trancoso, "Data-Driven Multithreading Using Conventional Microprocessors," *IEEE Trans. Parallel Distrib. Syst.*, vol. 17, no. 10, pp. 1176–1188, 2006.

[5] K. Stavrou, M. Nikolaides, D. Pavlou, S. Arandi, P. Evripidou, and P. Trancoso, "TFlux: A Portable Platform for Data-Driven Multithreading on Commodity Multicore Systems," in *ICPP '08: Proceedings of the 2008 37th International Conference on Parallel Processing*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 25–34.

[6] K. Stavrou, D. Pavlou, M. Nikolaides, P. Petrides, P. Evripidou, P. Trancoso, Z. Popovic, and R. Giorgi, "Programming abstractions and toolchain for dataflow multithreading architectures," in *ISPDC '09: Proceedings of the 2009 Eighth International Symposium on Parallel and Distributed Computing*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 107–114.

[7] Z. Budimlic, A. M. Chandramowlishwaran, K. Knobe, G. N. Lowney, V. Sarkar, and L. Treggiari, "Multi-core implementations of the concurrent collections programming model," in *CPC '09: 14th Workshop on Compilers for Parallel Computing*. Springer, 2009.

[8] P. Evripidou, "Thread Synchronization Unit (TSU): A Building Block for High Performance Computers," *In: Proceedings of the International Symposium on High Perfomance Computing, Fukuoka, Japan.*, 1997.

[9] C. Kyriacou, P. Evripidou, and P. Trancoso, "Cacheflow: A Short-Term Optimal Cache Management Policy for Data Driven Multithreading,," *Proc. EuroPar-04*, pp. 561–570, Aug. 2004.

[10] S. Arandi and P. Evripidou, "DDM-VMc:The Data-Driven Multithreading Virtual Machine on the Cell Processor ," Univerisity of Cyprus, Tech. Rep. TR-09-1, 2009.

[11] D. E. Culler and Arvind, "Resource requirements of dataflow programs," *SIGARCH Comput. Archit. News*, vol. 16, no. 2, pp. 141–150, 1988.

[12] C. A. Ruggiero and J. Sargeant, "Control of parallelism in the manchester dataflow machine," in *Proc. of a conference on Functional programming languages and computer architecture*. London, UK: Springer-Verlag, 1987, pp. 1–15.

[13] J. P. Perez, P. Bellens, R. M. Badia, and J. Labarta, "Cellss: making it easier to program the cell broadband engine processor," *IBM J. Res. Dev.*, vol. 51, no. 5, pp. 593–604, 2007.